Engineering of Safe Autonomous Vehicles through Seamless Integration of System Development and System Operation

D o c t o r a l T h e s i s
(D i s s e r t a t i o n)

to be awarded the degree of
Doctor rerum naturalium
(Dr. rer. nat.)

submitted by
Malte Mauritz
from Braunschweig

approved by the Faculty of Mathematics/Computer Science
and Mechanical Engineering,
Clausthal University of Technology

Date of oral examination
28th of June 2019

Dean
Prof. Dr.-Ing. Volker Wesling

Chairperson of the Board of Examiners
Prof. Dr.-Ing. Michael Prilla

Supervising tutor
Prof. Dr. Andreas Rausch

2. Reviewer
Prof. Dr. Falk Howar

Dissertation Technische Universität Clausthal, SSE-Dissertation 21, 2019

Abstract

Autonomous vehicles will share the road with human drivers within the next couple of years. This will revolutionize road traffic and provide a positive benefit for road safety, traffic density, emissions, and demographic changes.
One of the significant open challenges is the lack of established and cost-efficient verification and validation approaches for assuring the safety of autonomous vehicles. The general public and product liability regulations impose high standards on manufacturers regarding the safe operation of their autonomous vehicles. The vast number of real-world traffic situations have to be considered in the verification and validation. Today's conventional engineering methods are not adequate for providing such guarantees for autonomous vehicles in a cost-efficient way. One strategy for reducing the costs of quality assurance is transferring a significant part of the verification and validation from road tests to (system-level) simulations. The vast number and high complexity of real-world situations complicate the exhaustive verification of autonomous vehicles in simulations. It is not clear, how simulations address the vast number of real-world situations with sufficient realism and how their results transfer to the real road.
Extensive coverage of real-world situations in simulations requires the integration of development and operation. This thesis presents an engineering approach that integrates the development and operation of autonomous vehicles seamlessly using runtime monitoring. The runtime monitoring verifies if autonomous vehicles satisfy their requirements and operate within safe limits which have been verified in the simulations.
Safety of autonomous vehicles is subject to the scope of verified traffic situations in simulations. Systematic and comprehensive simulations support the improvement of autonomous vehicles and coverage of traffic situations. Results of the runtime monitoring during operation are transferred to the development for the verification of autonomous vehicles and their safe limits in simulations with additional traffic situations.
The incomplete verification of autonomous vehicles for the vast number of real-world traffic situations in simulations requires the validation of simulation results and additional monitoring in the real world. Results from simulations are transferred to the runtime monitoring during operation in the real world for validating the realism of the simulations and maintaining the vehicle safety in critical situations.
Vehicle data and real-world situations possess high complexities and, therefore, impact the complexity and efficiency of the verification in simulations. The runtime monitoring abstracts from internal data of autonomous vehicles and real-world situations in the evaluation by introducing an abstract semantic representation from natural language requirements.
A case study evaluates the engineering approach for an industrial lane change assistant and real-world traffic data recorded in road tests on German highways.

Acknowledgments

This thesis is the result of my research at the Institute for Applied Software System Engineering (IPSSE), TU Clausthal. I thank my supervisor Prof. Dr. Andreas Rausch for the opportunity to work at the IPSSE and mentoring me throughout this thesis. I am grateful for the valuable experiences during my time at the IPSSE. I want to thank Prof. Dr. Falk Howar for our collaboration in my research project, reviewing this thesis, and the possibility to continue my research at the Fraunhofer Institute for Software and Systems Engineering (ISST). The discussions and your feedback have significantly contributed to this thesis. I thank Prof. Dr. Ursula Goltz for the opportunity to begin my academic career at the Institute for Programming and Reactive System (IPS).
My years at the Institute for Applied Software System Engineering (IPSSE) have been an interesting and enjoyable experience. The discussion, criticism, suggestions, and feedback from my colleagues during this time have been a valuable input for me personally and this thesis. Therefore, I want to thank all my colleagues; Henrik Peters, Arthur Strasser, Peter Engel, Axel Grewe, Dr. Christoph Knieke, Benjamin Cool, Marco Körner, Dirk Herrling, Christian Ristig, Martin Vogel, Simone Dahms, Fadi Jabbour, Phillip Wolter, Steffen Kuepper, Adina Aniculaesei, Jörg Grieser, Leonard Scholz, Marco Kuhrmann, Thorben Knust. I further thank Dr. Sascha Lity, Dr. Lukas Märtin, Stephan Mennicke, Dr. Hauke Baller, Dr. Jens-Wolfhard Schicke-Uffmann, Benjamin Mensing, and Dr. Matthias Hagner of the (former) Institute for Programming and Reactive System (IPS) for two years with unforgettable memories. I also thank all members of the chair of Software System Engineering (SSE), Clausthal.
This thesis originates from my research in the research project Dependability Advanced Driver Assistant Systems (DADAS). I want to thank the project partners Volkswagen AG, Institute of Control Engineering (IFR), and Institute of Software Engineering and Automotive Informatics (ISF) of the TU Braunschweig for the fruitful collaboration. My thanks go to Prof Dr.-Ing. Markus Maurer, Prof Dr.-Ing. Ina Schaefer, Dr. Arne Bartels, Dr. Lutz Junge, Thomas Ruchatz, Jens Krause, Till Menzel, Dr. Fabian Schuldt, Dr. Simon Ulbrich, and Benjamin Schmidt.
Finally, I thank my parents, Christina and Bernd, for providing me with the opportunity and support to succeed through my years as a student and doctoral researcher. I also want to thank my girlfriend, Andrea, for enduring the highs and lows throughout these years and sacrificing on our shared time. I am grateful for all my friends who supported me during the years of preparing this thesis.

Malte Mauritz

Contents

List of Figures

List of Tables

Acronyms

ABS	anti-lock braking system
ACC	adaptive cruise control
ADAS	advanced driver assistance systems
ADTF	automotive data and time-triggered framework
ASIL	automotive safety integrity level
ASML	abstract state machine language
BASt	Federal Highway Research Institute (Bundesanstalt für Straßenwesen)
CAN	Controller Area Network
CNN	convolutional neural network
CPU	central processing unit
DCU	domain control unit
DGPS	differential global positioning system
DIL	driver-in-the-loop
DOF	degree of freedom
DSL	domain specific language
E/E	electrical and electronic
EBA	emergency brake assistant
ECU	electronic control unit
ESC	electronic stability control
FARS	fatality analysis reporting system
FMEA	failure mode and effect analysis
FPGA	field programmable gate array
FTA	fault tree analysis
GHG	greenhouse gases
GNSS	global navigation satellite system

GPS	global positioning system
HAZOP	hazard and operability study
HDD	hard disk drive
HDL	hardware description language
HIL	hardware-in-the-loop
HMI	human machine interface
IEEE	Institute of Electrical and Electronics Engineers
IIHS	Insurance Institute for Highway Safety
IPO	input-processing-output
LCA	lane change assistant
LHS	left-hand side
LIDAR	light detection and ranging
LKAS	lane keeping assist system
LTL	linear temporal logic
MDP	Markov decision process
MIL	model-in-the-loop
MLSL	multi-lane spatial logic
MOST	media oriented systems transport
MSC	message sequence chart
MTL	time metric temporal logic
NHTSA	National Highway Traffic Safety Administration
NLP	natural language processing
OS	operating system
PIL	processor-in-the-Loop
POMDP	partially observable Markov decision process
QoS	quality of service
RADAR	radio detection and ranging
RAM	random access memory
RHS	right-hand side
RSS	responsibility sensitive safety

SAA	sense and avoid
SAE	Society of Automotive Engineers
SAT	propositional satisfiability
SIL	software-in-the-loop
SoC	system-on-chip
SOTIF	safety of the intended functionality
SSTL	signal spatio-temporal logic
STL	signal temporal logic
SUT	system under test
TCM	transport class model
TCTL	timed computation tree logic
UML	unified modeling language
UTM	universal transverse Mercator
V2I	vehicle-to-infrastructure
V2V	vehicle-to-vehicle
V2X	vehicle-to-X
V&V	verification and validation
VAAFO	virtual assessment of automation in field operation
VDA	German Association of the Automotive Industry
VEHIL	vehicle-hardware-in-the-loop
VIL	vehicle-in-the-loop
VTD	Virtual Test Drive
VUT	vehicle under test
XIL	x-in-the-loop
XML	extensible markup language

1. Introduction

This thesis presents an engineering approach for autonomous vehicles as an extension of the current development process in the automotive industry. The motivation for this engineering approach and the contributions of this thesis are presented in this chapter.

1.1. Motivation

The road mobility sector is digitalized worldwide. While automation has been considered in the industries of aviation,e.g., drones, marine, e.g., submarines, and even to space exploration,e.g., Mars rovers, in the past decades, automation has been introduced to products of the automotive sector just recently (cf.[NL14; Wei+14]). *Automation* for the public road traffic is manifested as driver assistant systems, which support the driver in his driving task or even absolve the driver from the vehicle control and replace him [Fed17]. Today, advanced driver assistance systems (ADAS), e.g., adaptive cruise control (ACC) or lane keeping assist system (LKAS), are already deployed in vehicles (cf. Definition 2.5), but systems, which are able to accelerate resp. accelerated and steer the vehicle without any human input, are still under development [Fed17]. ADAS and systems for partial, high, and full automation of road vehicles are cumulated under the term *autonomous vehicle systems* in this thesis (cf. Definition 2.6). Nevertheless, there is a large gap between ADAS and full automated vehicles that are not recognized by the general public [NL14].
The Society of Automotive Engineers (SAE) classifies the automation of road vehicles into six levels — (0) no automation, (1) driver assistance, (2) partial automation, (3) conditional automation, (4) high automation, and (5) full automation (cf. Fig. 2.4). The capabilities of these systems to control longitudinal and lateral vehicle movement as well as to execute safety measures in critical traffic situations or system faults increase with each level of automation while the involvement of the human driver in the vehicle control and the supervision of the vehicle decreases. At automation level (0) – (2), the driver remains responsible for the monitoring of the vehicle's environment and initiation of necessary safety measure while autonomous vehicle systems with levels of automation (3) – (5) have to monitor their environment independently and perform necessary safety measure without input by the human driver. The common terms *self-driving vehicles*, *autonomous vehicles* refer to automation level (4) high automation and (5) full automation (cf. Section 2.1.4).
Highly and full automated vehicles have the potential to revolutionized road traffic and are envisaged to provide a positive benefit for social, economic, and environmental challenges (cf. [BR15; Fed17; NL14]):

Road Safety Autonomous vehicle systems are envisaged to reduce road accidents and fatalities on public roads and help to achieve the vision of "zero fatalities" (cf. [Eur11], [WH06]), if the highest road safety is maintained. In the year 2016, 34.439 crashes with over 37.000 fatalities happen in the United States[1] and 308.100 crashes with personal injuries led to 3.206 fatalities in Germany [Rad17].

88.1% of road accidents with personal injuries on German roads are subject to errors by human drivers [Rad17]. In the United States, it is estimated that around 93% or accidents are subject to human errors [NL14]. Drunk drivers account for 41% of all fatal crashes according to the NHTSA. Distracted drivers are responsible for 41% of fatal crashes, and fatigue drivers account for 2% of all fatal crashes (cf. [NL14]).

Autonomous vehicle systems are not subject to these human errors and, therefore, can mitigate a majority of these road accidents. Furthermore, autonomous vehicle systems might also outperform human driver in perception,e.g., blind spot detection, decision making, e.g., more accurate planning of complex driving maneuvers, and vehicle control, e.g., faster and precise steering and braking) in the majority of traffic situations [KP16].

The Insurance Institute for Highway Safety (IIHS) estimates that nearly a third of today's road accidents could be prevented if all of today's vehicles would be equipped with ADAS, e.g., dynamic brake support, forward collision warning, lane departure warning, and blind spot assists [NL14].

Road safety is the primary concern for autonomous vehicle and has to prioritized above all other challenges [Aga+16; Fed17].

Traffic Density The total annual mileage on German roads by 2020 is envisaged to increase by 21% in comparison to the annual mileage in the year 2002 [aca06]. The most substantial increase in annual mileage is predicted for the cargo transportation with 34%. The increasing road traffic volume will lead to more problems, e.g., traffic jams if the infrastructure is not extended. Autonomous vehicle systems can help optimize the use of existing infrastructure by improving the overall traffic flow and bypassing traffic jams on alternative routes resulting significant time savings and increasing comfort for passengers (cf. [BR15; NL14]).

Emissions Autonomous vehicle systems can help to further decrease the emission of carbon monoxide (CO), carbon dioxide (CO_2), and other greenhouse gases (GHG) and contribute to national and international climate strategies and targets (cf. [Cap+13]). Today, automobiles account for 20% of the total GHG emissions [BR15]. Inefficiencies in the driving behavior of human drivers further account for a part of this percentage in addition to the high emissions of combustion engines. The reduction of traffic accidents and the improvement of traffic flow by autonomous vehicle systems would directly impact the GHG emissions. Autonomous vehicle

[1] retrieved from the fatality analysis reporting system (FARS) of National Highway Traffic Safety Administration (NHTSA): https://www-fars.nhtsa.dot.gov/Main/index.aspx (Accessed: 12/06/2018)

systems with low automation level, e.g., ACC, can improve fuel economy by $4-10\%$ [NL14]. A higher saturation of road traffic by high automated vehicles would lead to a further reduction of GHG emissions.

Demographic Change Most industrial countries face the challenge of an increasing elderly society. Autonomous vehicle systems can help to support the mobility of elderly and disabled people. Deficiencies of these groups, e.g., impaired perception, can be compensated by autonomous vehicle systems. High automated vehicles would enable participation of, e.g., blind and disabled people in public road traffic because their high level of automation does not require any human control and supervision (cf. [Fed17; NL14]). The individual mobility of elderly and disabled people has a significant benefit for the integration of these people into society [Fed17].

In addition to the social and political driver for automated driving, also the technical components, e.g., sensor and actuator, as well as the necessary processing powers are available in large quantities with sufficient qualities in order to realize autonomous vehicle systems with increasing autonomy [BR15].
In recent years, major car manufacturers such as General Motors, Mercedes Benz, and Audi, and tech companies like Google, Uber, and Apple, have been racing towards autonomous road vehicles. The primary objective in this race seems to be the first to deploy a full automated production vehicle on the road [SSS17]. Some car manufacturers believe that full automated vehicles could exist by 2025 [NL14]. Towards full automated vehicles, car manufacturers are going to introduce autonomous vehicle systems with increasing levels of automation gradually over time [BR15]. The current generation of autonomous vehicle systems includes highway pilots which autonomously follow traffic on highways — including traffic jams. However, human drivers remain responsible for the safety of the vehicles and have to supervise the system and its environment. In the presence of critical situations, human drivers are required to intervene and mitigate the emerging risks.
All major car manufacturers and tech companies which work on autonomous vehicles have prototype vehicles with autonomous capabilities operating on public roads. Waymo has accumulated over 8.0 million miles on public roads with their fleet of over 80 autonomous prototype vehicles[2]. In the state California, United States, Waymo drove 352,544.6 miles autonomously on public roads in the year 2017. This mileage is the most of all companies to test autonomous vehicles on public roads, e.g., Bosch accumulated 6,305 autonomous miles, General Motors Cruise drove 13,1677 miles autonomously, Nissan tested on 5007 miles, and Mercedes Benz accumulated 1088 miles. However, these companies may accumulate additional test miles in tests in other states or countries. The mileage in California are tracked by disengagement reports of the Department of Motor Vehicle of the State California[3].

[2] https://www.theverge.com/2018/7/20/17595968/waymo-self-driving-cars-8-million-miles-testing (Accessed: 12/06/2018)

[3] https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/disengagement_report_2017 (Accessed: 12/06/2018)

Autonomous vehicle systems impose both enormous potential benefits and enormous potential risks [KP16; Wei+14]. Autonomous vehicle systems are commonly classified as *safety-critical* because their failures impose the possibility for death and injuries of humans [BV10]. While car manufacturers and tech companies have made significant progress in the development of the autonomous functionality for road vehicles, the risks and potential faults of autonomous vehicle systems have not been equally addressed until recently [NL14]. The disengagement reports of the Department of Motor Vehicle of the State California disclose that all companies encountered critical and unsafe situations with their autonomous prototype vehicles on public roads, i.e., Waymo, had 63 disengagements, Bosch had 595 disengagements, GM Cruise accumulated 105 disengagements, Nissan counted 24 disengagements, and Mercedes Benz had 840 disengagements in the year 2017.

Several prototype vehicles of different operators have been involved in accidents with fatalities. In Mai 2016, a driver of a Tesla Model S has been killed when its car with engaged autopilot system drove into the white truck trailer which was crossing the highway[4]. In March 2018, a prototype vehicle of Uber operating in self-driving mode killed a crossing pedestrian at night in Tempe, Arizona[5]. At the same time, a Tesla Model X with engaged autopilot system drove into a concrete lane divider and burst into flames[6]. The driver of the Tesla later died at the hospital.

Road safety is the primary concern for the introduction of autonomous vehicle systems in road vehicles [Fed17]. Autonomous vehicle systems must be sufficiently safe. Car manufacturers and tech companies must minimize any unintended consequences and imminent risks of this technology, i.e., collisions, in order for autonomous vehicle systems to have a positive impact on mobility, safety, and time consumption [NL14].

Nevertheless, vehicles with autonomous vehicle systems will still be involved in road collisions. A 100% safety on roads is doubtful to be achieved even with autonomous vehicle systems [SSS17; Sti13]. As collisions with autonomous vehicles cannot be completely ruled out, residual risks for objects and persons in the vicinity of autonomous vehicle systems are imminent. The safety risks of autonomous vehicle systems is particularly present in traffic with different levels of automation, in combination with human controlled vehicles and other traffic participants, i.e., bicycles and pedestrians, or extreme weather conditions (cf. [Fed17]). In some of these situations, human drivers are challenged. Autonomous vehicle systems might perform worse than human drivers [KP16].

Autonomous vehicle systems have to provide a balance of risks for the overall road safety. The performance of autonomous vehicle systems has to least match the driving performance of human drivers [BR15; Fed17]. In critical traffic situations, autonomous vehicle systems must be able to mitigate the risks by autonomously performing emergency maneuvers without input from the human driver [BR15]. Otherwise, this technology will not be accepted by the general public [Fed17; NL14; SSS17]. The fatal collisions of

[4]https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk (Accessed: 12/06/2018)

[5]https://www.theverge.com/2018/5/24/17388696/uber-self-driving-crash-ntsb-report (Accessed: 12/06/2018)

[6]https://www.wired.com/story/tesla-autopilot-self-driving-crash-california (accessed: 12/06/2018)

autonomous prototype vehicles with fatalities have arisen first doubts about autonomous technology in public road traffic.
The current approach to developed autonomous vehicle systems based on based geometric signal data from sensor measurement lacks the interpretability and explainability of any system actions (cf. [SSS17]). The general public expects non-technical explanations for actions by autonomous vehicle systems leading to accidents. A formal semantic model for the reasoning about behavior and safety of autonomous vehicle systems is required. The planning of autonomous vehicle systems is unlike scale up exponentially with time and number of dynamic object in their environments. Current development approaches for autonomous vehicle systems — including the safety standard ISO 26262— does not provide any semantic model. [SSS17]
Questions about the liability for the operation of autonomous vehicle systems arise because collisions cannot be completely mitigated. With increasing autonomy of autonomous vehicle systems, the responsibility for the correctness and safety of these systems during operation on public roads shifts from human drivers to car manufacturers and public authorities. Without the human driver in supervision, the product and producer liability (cf. ProdHaftG §1, BGB §823 I, BGB §433) define manufacturers to be liable for any damage by their autonomous vehicle systems (cf. [Wei+14]). Car manufacturers, i.e., Daimler AG and Volkswagen AG, plan to maintain high safety standards for their autonomous vehicle systems by introducing increasing automation capabilities for their autonomous vehicle systems gradually over time [BR15].
Car manufacturers are obliged to ensure the safety of their autonomous vehicle systems in the context of the current technology state by continuously monitoring, testing, and optimizing their autonomous vehicle systems [Fed17; NL14; Sch12; Wei+14]. The safety of autonomous vehicle systems consists of two essential aspects (cf. [Sti13; Wei+14]):

Loss of function: The loss of function can arise if technical components, i.e., electronic control units (ECUs) or sensors, fail. Autonomous vehicle systems must be designed redundantly with equivalent fall-back components in order to compensate such loss of function. For full automated vehicles, the autonomous vehicle systems have to transfer the vehicle into a safe state in which any danger is mitigated. For lower levels of automation, autonomous vehicle systems can hand over the control of the vehicle back to the human driver but must safely operate autonomously for the time of the handover. [Sti13]

Loss of integrity: The loss of integrity may arise as a shortcoming by the nominal functionality of autonomous vehicle systems even if no system fault is present. For example, the perception of autonomous vehicle systems may fail to correctly recognize a real-world object in the path of the vehicle. As a result, the decision by the autonomous vehicle systems in this situation could be faulty and unsafe, i.e., driving onto this object — because the internal environment representation does not correctly model the real world. [Sti13]

The *loss of function* is addressed by the safety standard ISO 26262 [Sti13]. The safety standard ISO 26262 represents the specific interpretation of the common norm IEC 61508

for *functional safety* for the automotive domain. The norm ISO 26262 defines a safety life cycle for the *functional safety* of electrical and electronic (E/E) systems in automobiles up to 3.5t. For the mitigation of potential *loss of function* due to faults of technical systems components, methods, activities, and work items are defined throughout the development and operation of E/E systems. Safety requirements define measures for the potential *loss of function* of software and hardware components based on a risk assessment of these components. These safety requirements have to be considered throughout the development, verification, and validation of the vehicle and its systems. [Sti13; Wei+14] With the increasing level of automation, the safety requirements for autonomous vehicle systems have to be more restricting in order to maintain the acceptable measure of risks [NL14].
However, the safety standard ISO 26262 is not sufficient for developing safe autonomous vehicle systems because the safety standard ISO 26262 does not sufficiently address the *loss of integrity*. For the safety standard ISO 26262, vehicle systems either meet their requirements and any risks by the systems is excluded or the vehicle systems operate outside their specifications due to failures of system components [Wei+14]. The safety standard ISO 26262 does explicitly exclude the performance of the intended nominal system functionality and considers the human driver as permanent fall-back [Sti13]. Autonomous vehicle systems are unlikely to be specified and tested to the extent that all situation-critical maneuvers reside outside their specifications [Wei+14].
A new safety standard ISO/PRF PAS 21448 — safety of the intended functionality (SOTIF) — is currently under development in order to address the *loss of integrity* for partially automated vehicles and complement the norm safety standard ISO 26262. The norm SOTIF addresses the following aspects in the development of autonomous vehicle systems up to partially automated vehicles:

- pattern for the system architecture of autonomous vehicle systems,
- evaluation of risks by the intended functionality, which is not addressed by the safety standard ISO 26262,
- identification of scenarios and events and evaluation of emerging safety risks,
- verification, validation, and risks reduction for the intended functionality,
- definition of criteria for the approval of autonomous vehicle systems in production vehicles.

The definition of the standard ISO/PRF PAS 21448 — SOTIF — correlates to current research projects, e.g., the PEGASUS project (cf. [Win+18]). The PEGASUS project aims to research generally accepted quality criteria, tools, and methods which are representative for the release of highly-automated vehicle functions. The quality criteria include the investigation and definition of relevant and representative scenarios and situations in which these functions have to operate safely.
The safety standard ISO/PRF PAS 21448 and the PEGASUS project try to address the question; “When autonomous vehicle systems are sufficiently safe?”. Until today, neither

car manufacturers, tech companies, nor public authorities have defined any measure for the acceptable risks of the autonomous vehicle system in public road traffic. However, a definition of the acceptable risks is inevitable because car manufacturers and tech companies require legal certainty for their development of autonomous vehicle systems [BR15; Sti13]. A common hypothesis is that the risk for fatalities by autonomous vehicle system in public road traffic must not be greater than the risks for fatalities by human drivers [Win15].
The approval of autonomous vehicle systems for the public road traffic resides with public authorities, but car manufacturers have to provide sufficient evidence about the safety of their autonomous vehicle systems [Fed17; Wei+14]. Car manufacturers and tech companies have to verify and validate that their autonomous vehicle systems remain within the accepted residual risks for a positive impact on public road traffic [Kna+17b; NL14; Wei+14].
Besides uncertainties about the accepted residual risks of autonomous vehicle systems, one other challenge in the development of autonomous vehicle systems is the current lack of established and cost-efficient approaches for assuring the safety of autonomous vehicle systems towards full automated driving [Den+14; Kna+17b; Sti13; Win15]. Until today, no valid concept for verification and validation of autonomous vehicle systems is known [Wei+14]. Today's conventional engineering methods in the automotive domain are incapable of providing such safety guarantees for autonomous vehicle systems.
The common practice for the safety verification and validation of E/E systems in the automotive industry is the testing with prototype vehicles on public roads. The safety of E/E system in these tests is quantified by the failure rates of these systems over the driven mileage. This practice has been successfully applied for driver assistance systems, e.g., the emergency brake assistant (EBA), but has little to no significance for the autonomous vehicle system with environment perception and autonomous decision making in public road traffic (cf. [SSS17; Win15]). The safety evaluation for the EBA focused only on *false positives* but for autonomous vehicle systems also *false negatives* have to be evaluated [Sti13]. Prof. Winner et al. estimate that the required mileage for the verification and validation of autonomous vehicle systems on public roads succeeds 240m km [Win15]. The authors of [WW17] calculate a necessary mileage of 6.6 billion kilometers while the authors of [KP16] estimate 8.8 billion miles. Such mileages are not feasible for car manufactures and tech companies under reasonable time and costs [KP16; Sti13; WW16]. Winner et al. call this problem the *approval-trap* (cf. [Win15; WW16]). Car manufacturers and tech companies are required to reverify and revalidate the performance of their autonomous vehicle systems for at least a third of the initial test effort following each change of these systems [KP16; Win15]. The reproducibility of real-world tests is limited — if not impossible — due to a large number of factors and variations in public road traffic — some of them are not even unknown [GRS14; HK16].
The introduction of learning and self-learning algorithms for autonomous vehicle systems impose another challenge for the verification and validation of autonomous vehicle systems. Learning algorithms, e.g., object identification by convolutional neural networks (CNNs), are trained in the development while self-learning algorithms, e.g., the automatic adaption of vehicle dynamics, update their knowledge base continuously during operation

[Fed17]. Even though CNNs commonly outperform human drivers in the perception and classification of environment objects, the internal learning of these nets and its semantics are not yet well understood (cf. [Mar18; NYC15]). Insufficiencies, i.e., adverse side effects, rewards hacking, or sensitiveness to environment differences by learning and self-learning algorithms impose a significant threat to the safety of autonomous vehicle systems and have to be eliminated (cf. [Amo+16]). However, learning and self-learning algorithms currently lack the necessary sound verification, and validation approaches [Gha+18; GP17; Kna+17b]. A separation of self-learning systems and safety-critical vehicle functions is not feasible (cf. [Fed17]). For example, the environment perception is a fundamental and safety-critical component of autonomous vehicle systems and has a direct impact on all other system functions, e.g., situation assessment, decision making, and vehicle control.

New approaches for the verification, validation, and risk assessment of autonomous vehicle systems have to be developed in order to ensure the safety and positive impact of autonomous vehicle systems [Gas12; KP16; Sti13; Wei+14; Win15; WW16]. These approaches have to consider with the vast quantity of diverge traffic situations and the vast set of possible behaviors by the autonomous vehicle systems [Aga+16; BJ05; Wei+14].

Critical traffic situations for the safety of autonomous vehicle systems are traffic situations which have not been sufficiently considered in the development of the autonomous vehicle systems — including traffic situations which are easily handled by human drivers (cf. [Sti13]). Any approach must be able to identify and document these critical but seldom traffic situations (cf. [Aga+16; BR15; Sti13]). However, no approach is currently able to determine the theoretical completeness of critical traffic situations in order to guarantee the safety of autonomous vehicle systems [Wei+14] soundly.

One strategy in the automotive domain for reducing the costs of quality assurance is the application of the structured testing as well as transferring a significant part of the testing effort from tests on public roads to (system-level) simulations [BR15; Kna+17a; Kna+17b]. For these simulations, software and hardware components of the autonomous vehicle systems are integrated into a simulation framework. The simulation frameworks simulate virtual worlds and generate the test input data for the autonomous vehicle systems. The behavior of the autonomous vehicle systems can be transferred back to the simulation framework and impact the state of the virtual world. In comparison to real-world testing, simulations offer an improved reproducibility, flexibility and reduction of costs and time for the verification and validation of autonomous vehicle systems [HK16]. Nevertheless, simulations are unlikely to verify and validate all possible variants of traffic situations and singlehandedly account for the necessary distance of million miles which are statistically required for the approval of autonomous vehicle systems [Win15].

Simulations have to sufficiently cover relevant and realistic traffic situations for valid statements about the safety of autonomous vehicle systems. Realistic traffic situations represent real-world traffic situations which autonomous vehicle systems are likely to encounter in the real world. A systematic approach for systematically identifying traffic situations does not yet exist. (cf. [Sch+13]) The number of traffic situations in test sets should be as minimal as possible in order to limit the costs of the safety assurance in

general and in systems simulations particular. Neither car manufacturers nor public and legal authorities have yet defined any requirements for a sufficient but minimal set of traffic situations for the verification and validation of autonomous vehicle systems [Aga+16; Wei+14; Win15]
Any results of the system simulations must be employable for the operation of autonomous vehicle systems in public road traffic. The behavior of autonomous vehicle systems in simulations and on the road in the real world have to be comparable. Otherwise, system simulations no impact on the verification and validation of autonomous vehicle systems. As system simulations will not verify and validate all possible real-world traffic situations which autonomous vehicle systems may encounter in the real world, the impact for the safety of autonomous vehicle systems by the system simulations should be quantified (cf. [Wei+14]). However, until today no metric has been defined for the performance of autonomous vehicle system based on the set of covered traffic situations (cf. [Win15]). Nevertheless, intelligent distribution of the verification and validation between system simulations and real-world tests will have a significant impact on the time-to-market of safe autonomous vehicles [Sti13].
The introduction of autonomous vehicle systems towards full automated vehicles requires a consistent understanding required vehicle functions of the different automation levels as well as the solution of the social, legal, and technical challenges [BR15]. This work is envisaged to provide contributions to the verification and validation of autonomous vehicle systems by addressing the shortcomings in the identification of relevant and realistic scenarios and the transferability of simulation results. The contributions of this work are described in the following section.

1.2. Thesis Contributions

The research in this thesis makes the following contributions:

A thorough analysis of the current development process in the automotive domain concerning the safety of autonomous vehicle systems.

In this thesis, the current development practice in the automotive domain for autonomous vehicle systems is presented on the example of a highway pilot with a lane change assistant — from requirements engineering to verification and validation. This thesis analyses the development process and discloses its problems regarding the safety of autonomous vehicle systems.

Improving the safety assurance in the development process of autonomous vehicle systems by integrating runtime monitoring data in simulations and during operation in the real world.

This thesis proposes an extension for the current development process in the automotive domain by incorporating runtime monitoring into simulations of the system verification and during operation in the real world in order to address identified problems for the development of safe autonomous vehicle systems. The proposed approach aims to improve the safety of these autonomous vehicle

systems iteratively. The gap between simulations in the system verification and the operation of autonomous vehicle systems in the real world is bridged by transferring runtime monitoring results between the simulations and the operation.
Three additional contributions are aligned with the proposed engineering approach:

A component architecture for runtime monitoring of autonomous vehicle systems and their environments at an abstract level.

A component architecture for the runtime monitoring of autonomous vehicle systems is defined in this thesis which abstracts from the concrete signals of the autonomous vehicle systems and monitors system properties on the abstract level of the system requirements. The runtime monitoring verifies the correctness and safety of autonomous vehicle systems in each encountered situation and records resp. compares the encountered situation with the known situations from the simulations of the system verification. The runtime monitoring accesses signal values of the autonomous vehicle systems and transforms them into an abstract representation of situations.

The definition of safety-relevant system properties from system requirements on an abstract level.

For the runtime monitoring of autonomous vehicle systems, a pattern-based approach is introduced in this thesis for the definition of system properties from the requirements of the autonomous vehicle systems. The transformation by the pattern-based approach results in a formal definition of an abstract representation for the states of autonomous vehicle systems and their environment situations. The runtime monitoring uses this abstract representation for the verification of the autonomous vehicle systems and the recognition of environmental situations.

A framework for estimating and improvement of the impact and scope by the system verification for the operation of autonomous vehicles systems during operation.

During operation in the real world, the runtime monitoring framework can identify situations for which the autonomous vehicle systems have not been sufficiently verified in simulations of the system verification. This information enables to determine the impact and scope of the system verification based on the ration of safe and critical situations during the operation of autonomous vehicles systems. Critical situations which have been recorded during operation are used for the improvement of the autonomous vehicle systems and the definition of additional simulations in the system verification. Therefore, the proposed engineering approach continuously improves the impact and scope of the system verification as well as the correctness and safety of autonomous vehicle systems.

The definition of test scenarios and test cases for the system verification from runtime monitoring results during operation in the real world.

This thesis describes the process of defining new simulations for the system verification from results of the runtime monitoring of autonomous vehicle systems

during operation in the real world. The runtime monitoring records environment situations in which the autonomous vehicle systems violates the system properties or which have not been encountered in the simulations of the system verification. The recorded situations are analyzed, and test scenarios are defined based on changes between recorded situations. These test scenarios describe changes in the scenery and behavior of dynamic objects, e.g., vehicles, in the abstract representation of the runtime monitoring. Concrete test cases finalize abstract parameters of test scenarios for the simulations in the system verification of autonomous vehicle systems.

The following section presents an outline of this thesis.

1.3. Outline

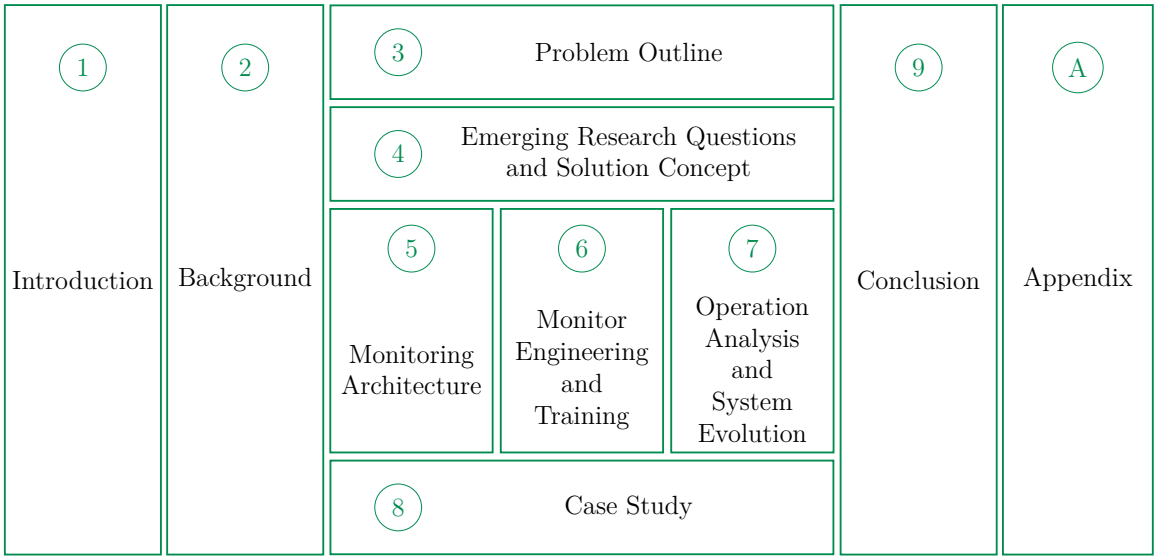


Figure 1.1.: Overview about the content of this thesis.

The remainder of this thesis is organized as shown in Fig. 1.1:

Chapter 1 (this chapter) gives an introduction to the problems for the verification and validation of autonomous vehicle systems as well as the contributions of this work.

Chapter 2 summaries the necessary background and knowledge for the following chapters of this thesis.

Chapter 3 presents the current development process in the automotive domain on a highway pilot with integrated lane change assistant and discloses the problems of this process for the development of autonomous vehicle systems.

Chapter 4 gives an overview over related academic and industrial work, the emerging *research questions*, and the *solution concept* of this thesis. The *solution concept*

constitutes a combination of runtime monitoring of autonomous vehicle system in simulations of the system verification and during operation in the real world.

Chapter 5 describes the architecture of the runtime monitoring framework and its integration with the autonomous vehicle systems.

Chapter 6 illustrates the definition and implementation of the runtime monitoring framework based on requirements of autonomous vehicle systems and its training in simulations of the system verification.

Chapter 7 describes the usage of runtime monitoring results during operation of autonomous vehicle systems in the real world as well as the usage of results from the runtime monitoring during operation for the improvement of the autonomous vehicle systems and their verification in simulation in further development iterations.

Chapter 8 presents the evaluation of the proposed engineering approach for the lane change assistant in chapter 3 — including the result of the runtime monitoring in simulations of the system verification and during operation for recordings of real-world test drives.

Chapter 9 summarizes this thesis and concludes this thesis with a discussion of the proposed engineering approach and an outlook on future work.

Appendix A provides additional information about data structures and test scenarios which have been implemented within the evaluation of the engineering approach on the lane change assistant.

2. Background

The following section introduces basic concepts about autonomous vehicle systems, simulation-based testing, runtime monitoring, and typed first-order logic for the understanding of content in the following chapters.

2.1. Autonomous Vehicle Systems

The advent of autonomous systems in recent history started in 1940 when Norbert Wiener reasoned about the similarity of intelligent behavior of servomechanisms for anti-aircraft guns with the nominal and anomalous operation of biologic systems (cf. [HB82]). Wiener's work led to the formalization of a theory of *feedback control* and its generalization to human biologic systems (cf. [WS05]). The first generation of autonomous systems emerged from this work combining simple sensors and effectors with analog control electronics.
The advent of digital control electronics in the 1970s and advances in automated perception and cognition led to *autonomous systems* that could plan and execute complex operations with minor to no human intervention. With increasing progress and reduction of costs for sensors, actuators, and processors, more sophisticated autonomous systems emerged in various domains, e.g., avionics, space, maritime, and automotive (cf. [WS05]).
Before the introduction and definition of autonomous vehicle systems in the automotive domain, common terminology and definitions for (autonomous) systems, in general, are introduced in the following sections.

2.1.1. Definition: System

Prior to the definition of *autonomous vehicle systems* (cf. Definition 2.6), basic terminology has to be introduced. A *system* is universally defined as a group or collection of interrelated *entities*, e.g.„ people or machines, for the accomplishment of a common objective (cf. [Ban+05; LK97; Sha98]). An *entity* is an object of interest in a system. Each entity has a set of *attributes* which characterize the properties of this entity [LK97]. A *subsystem* is a part of a system consisting of software components and hardware components.

Definition 2.1 (System)**.** *"A system is defined as a group of objects that are joined together in some regular interaction or interdependence toward the accomplishment of some purpose".[Ban+05]*

Systems are commonly categorized into two types; *discrete* or *continuous* systems. State of systems are characterized by assignments to their *state variables* which are necessary to describe the system concerning its objective. State variables of discrete systems change instantaneously at specific discrete points in time, while state variable of continuous systems changes overtime continuously (cf. [Ban+05; LK97]).
Events outside of the system often influence systems. "An *event* is defined as an instantaneous occurrence that might change the state of the system" [Ban+05]. It is essential to decide on the *boundary* of the system in order to distinguish between *endogenous* events within the system and *exogenous* events occurring in the *environment* of the system. The *system boundary* is individually defined in the system analysis for each system under consideration of its context and purpose (mission) (cf. [Ban+05]).

2.1.2. Definition: Autonomous System

Autonomous systems tend to be more flexible than traditional systems because *autonomous systems* commonly operate in dynamic and complex environments with a large set of circumstances, of which some may be unknown at design time of these systems. Autonomous systems must be able to adapt to these circumstances resulting in a much more considerable variation of system behavior (cf. [BJ05]).
The necessity for autonomous systems, becomes obvious, when these systems (cf. [Den+14; FDW13; HMA03])

- are deployed in remote environments where direct control is infeasible;
- are deployed in hostile environments where it is too dangerous for humans to assess the possibilities;
- involve actions that are too lengthy or frequent for humans to conduct successfully; or
- need to react more rapidly than humans can.

In such settings, longer periods without interventions of human operators are desirable or inevitable (cf. [Den+14]).
Autonomous system offer benefits for the *safety* of human life and the mission of these systems (cf Definition 2.2). *Autonomy* is an emergent system property and can enhance the achievement of the mission, task, goals, as well their accuracy and repeatability in time and space under the saving of time, space, and material (cf. [Con+06; Dur05; HMA03]). It may be even cheaper to use *autonomous systems* in comparison to the training, monitoring, safety, medical support, legal oversight which humans may require for the same task (cf. [Den+14]).

Definition 2.2 (Safety)**.** *"The ability of a system not to cause danger to persons or equipment or the environment" [Ise06b].*

The term *Autonomy* is defined in the Merriam-Webster Dictionary as "the quality or state of being self-governing"[Mer18] and by the Oxford dictionary as the "right or condition of self-government"[Ste10]. However, in the field of autonomous vehicles *autonomy* is commonly related to something more synonymous with "independence" or "intelligence"[Con+06].
Stockton et al. propose the following definition for an autonomous system: "An autonomous [(sub-)] system is one that makes and executes a decision to achieve a goal without full, direct human control"[Con+06]. Watson and Scheidt call systems "autonomous" if systems can change their behavior in response to unanticipated endogenous and exogenous events during operation [WS05]. Schuhmann requires *autonomous systems* to execute a number of necessary steps without human intervention in order to achieve a given goal (cf. [SV06]).

Definition 2.3 (Autonomous System)**.** *"An* autonomous [(sub-)] system *is one that makes and executes a decision to achieve a goal without full, direct human control"[Con+06]*

Another terminology used in conjunction with autonomous systems is *automatic*. The distinction between *automatic* and *autonomous* systems is not commonly defined and widely discussed in academic literature (cf. [Con+06]). Some publications distinct *autonomous* and *automatic system* based on the decision inputs while other publications distinguish them based on the possible decision state space. However, counterexamples exists for either notion. Based on Definition 2.3, *automatic systems* are depicted as a subset of *autonomous systems* in this thesis (cf. [Con+06]). This notion is consistent with the definition of automation levels for autonomous vehicle systems by the SAE (cf. Section 2.1.4).
In this thesis, *autonomous systems* only encompass systems which interact with an open physical world — the real world — for the execution of their complex mission (cf. [WS05]). Such open physical worlds are "highly" dynamic with a large number of interactions, relationships, and moving objects. "Mobile" software entities, like viruses, daemons, or agents, are not considered in this thesis, even though these entities may operate with minor to no direct human interaction.
The complex behavior of autonomous systems is not a product of the complex systems themselves but rather the reflection of their complex environments (cf. [Bro86]). Autonomous systems are widely expected to operate in the same physical world as humans where they have to achieve multiple goals simultaneously — of which some may be conflicting — with none to a limited number of interventions by human drivers. They have tog. The systems are responsible for high-level goals, e.g., navigation, as well as consider necessary low-level goals, e.g., collision-free movement. The relative importance of concurrent goals is subject to the current system context.
For maintaining their high and low-level goals, autonomous systems are required to

- continuously perceive their environments and maintain an internal representation of these environments (cf. [Bro86]), and

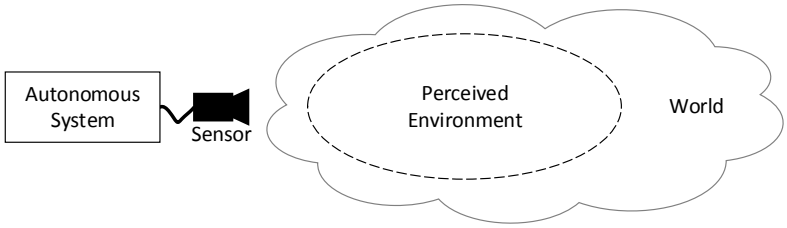


Figure 2.1.: Pictograph of the relation between real world and the system's perceived world.

- adapt their behavior in reaction to perceived changes in their internal states and the physical world (cf. [Con+06]).

For this behavior, decision making from the domain of artificial intelligence is combined with the real-time control used in robotics within autonomous systems (cf. [Con+06]). Environments which autonomous systems perceive represent only small portions of the real world (cf. Fig. 2.1). Autonomous systems are unlikely to encounter all possible traffic situations in the real world. Furthermore, novel situations emerge continuously as new objects, e.g., vehicles, are continuously introduced. Therefore, a set of situations which autonomous system encounter in the real world represents a subset of all possible real-world situations.
Traditional systems have been designed and analyzed under the *closed-world assumption*; that the complete system environments and the interactions of these systems with their environments can be specified entirely in the system development (design time). The *closed-world assumption* is not applicable to autonomous systems operating in open physical worlds, like the real world, due to the complexity of these worlds. Some aspects of the system environment are only known during operation (runtime). Autonomous systems operating in open physical worlds have to be defined and analyzed under the *open-world assumption*; that the environment of these autonomous systems cannot be specified entirely in the developments.

Definition 2.4 (Open-World Assumption)**.** *The* open-world assumption *for the development of (autonomous) systems assumes that the environment of these (autonomous) systems cannot be specified entirely in the development (design time). Some aspects of the system environment are only known during operation (runtime). The* open-world assumption *represents the antonym to the* closed-world assumption*, which assumes that environments of systems can be specified in their full extends in the system development.*

Autonomous systems have moved beyond the usage in industrial production and military, and are being deployed in home, health-care scenarios, and in automated vehicles. Autonomous systems for the autonomous driving of vehicles in public traffic without human input are addressed in the following section.

2.1.3. Definition: Autonomous Vehicle System

In the automotive domain, autonomous systems have their origin in early driver assistance systems, e.g., anti-lock braking system (ABS) and electronic stability control (ESC). Bosch introduced ABS in 1978. These systems solely measure internal vehicle parameters, e.g., difference between tire rotations, by *proprioceptive* sensors in order to support the driver in the control vehicle. The traditional assistance systems have evolved into ADAS. ADAS incorporate *exteroceptive* sensors, e.g., cameras and light detection and ranging (LIDAR) sensors, for the perception of the vehicle environment (cf. [Don+07]). Originating from ADAS, e.g., ACC, the ADAS have been further improved until today's system, e.g., the Audi AI traffic jam pilot (cf. [HBS18]), and will be further improved near future towards the vision of fully autonomous vehicles. Autonomous vehicles have been academically and industrially researched since the late 1980s.[Ben+14]

Definition 2.5 (Advanced Driver Assistance System)**.** *In contrast to traditional driver assistance systems, like ABS and ESC, advanced driver assistance systems incorporate signal data from exteroceptive sensors in addition to proprioceptive sensors into the processing of complex maneuvers for the control of vehicles (cf. [Don+07]).*

This thesis is particularly interested in ADAS and other vehicle systems which take over the vehicle control from the human driver and autonomously operate in the real world without or with minor input from the human driver. The vehicle control by these systems can be limited in time and space as well as terminate by the human driver (cf. [Win15]). These autonomous systems are cumulated in this thesis under the term *autonomous vehicle systems*. The definition *autonomous vehicle system* in this thesis corresponds to definition *driving automation system* by the SAE (cf. [SAE18]).

Definition 2.6 (Autonomous Vehicle System)**.** *The term "autonomous vehicle system" describes the set of systems in road vehicles which either support the driver in its task to control the vehicle in traffic or even replace him completely.*

The highest degree of automation for road vehicles by autonomous vehicle systems results in *full automated vehicles* or *autonomous vehicles* which can drive in any traffic conditions on public roads without human intervention. Autonomous vehicles are also depicted as *autonomous mobile robots* or *autonomous driving robots*. A detailed description of automation levels for autonomous vehicle systems is given in Section 2.1.4.
Berns and Puttkamer define six essential features for autonomous mobile robots operating (moving) in open-physical worlds (cf. [BP09]). These features equally apply to autonomous vehicle systems:

Mobility: The ability to move to specific positions in the world.

Adaptivity to Unknown Situations: The highly dynamic of open-physical worlds will confront robots (vehicles) with situations which have not been specified in their development. Therefore, adaptivity is a key feature.

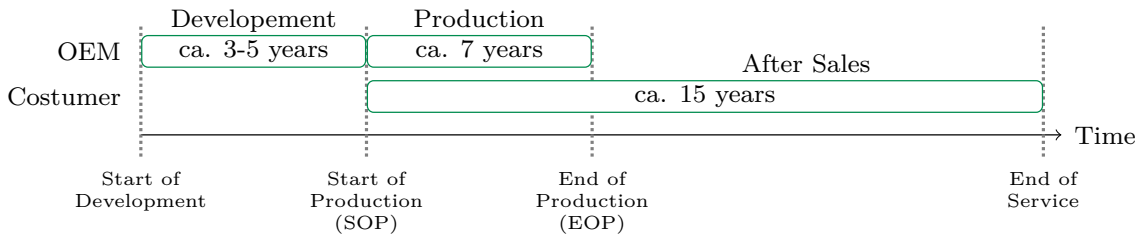


Figure 2.2.: Life cycle of vehicle and their system [SZ13].

Perception of Environment: For navigation and the fulfillment of their mission, robots (vehicles) have to retrieve information about their environments.

Knowledge Acquisition: The incomplete specified model of operational environment requires robots (vehicles) to acquire new knowledge while operating.

Safety: Robots (vehicles) have to ensure the safety of themselves and their environments. They must not damage themselves, any objects, or hurt any humans.

Real-Time: Robots (vehicles) must cope with hard real-time requirements.

These features have to be guaranteed for the complete life cycle of autonomous vehicle systems beyond their development and production. The life cycle of autonomous vehicle systems corresponds to the life cycle of vehicles which encompass up to or even over 20 years with the majority of time being the operation in production vehicles on public roads (cf. Fig. 2.2). This life-span has to be considered in particular in the context of safety analysis for autonomous vehicle systems. Car manufacturers are liable for any damage or injuries by the autonomous operation of their autonomous vehicle systems. While these systems operate in production vehicles, the possibilities for modifying systems and resolving of emerging system failures are limited.
The following section introduces the taxonomy of autonomy in the automotive domain.

2.1.4. Taxonomy of Autonomous Vehicle Systems

The taxonomy of autonomous vehicle systems in this thesis has two dimensions. In Section 2.1.4.1 introduces the three layers of the driving task by Donges et al. (cf. [Don82]). The categorization of automation levels for autonomous vehicle systems is introduced in Section 2.1.4.2.

2.1.4.1. Activities in the Driving Tasks

Tasks performed while driving road vehicles can be classified into one of three categories — primary, secondary and tertiary (cf. [Bub02]):

- The *primary driving task* addresses the control of the vehicle in traffic and can be separated into three actions — navigation, guidance, and stabilization.

- *Secondary driving tasks* encompasses all actions to configure the required operating mode of the vehicle, e.g., gear selection, activation of direction indicators, or windscreen wipers.

- Actions, which are related to modifying the inside ambient of vehicles, are summarized as *tertiary driving tasks*.

The SAE defines the *driving task* as "all of the real-time operational and tactical functions required to operate a vehicle in on-road traffic, excluding the strategic functions such as trip scheduling and selection of destinations and waypoints."[SAE14]. This definition includes (cf. [SAE14]):

- Lateral motion control via steering (operational);

- Longitudinal motion control via acceleration and deceleration (operational);

- Monitoring the driving environment via object and event detection, recognition, classification, and response preparation (operational and tactical);

- Object and event response execution (operational and tactical);

- Maneuver planning (tactical); and

- Enhancing conspicuity by lighting, signaling and gesturing. (tactical).

This definition by the SAE explicitly excludes the navigation — "trip scheduling and selection of destinations and waypoints" — from the *driving task* (cf. [SAE14; SAE18]). This thesis includes the navigation into the definition of the *driving task* and, therefore, follows the definition by Donges et al. (cf. [Don82]); The *driving task* encompass all activities related to the primary task of controlling the vehicle on public roads in traffic.

Definition 2.7 (Driving Task)**.** *The* driving task *encompass all activities related to the primary task of controlling the vehicle on public roads in traffic. These activities can be separated into a hierarchical model of three actions — navigation, guidance, and stabilization.*

Figure 2.3 displays the three layered model for the *driving task* by Donges [Don82]. The decomposition of the *driving task* into three hierarchical layers has been considered in several system architectures for autonomous vehicle systems in the past years (cf [Bau+12; Ber10; Bon+96; Mat15; Mau00b; Not14])
At the top layer, the *navigation* addresses the selection of an appropriate route, which leads from the starting point to the desired goal destination. Navigation systems consider the read network (cf. Fig. 2.3) for the calculation of the route under consideration of constraints, e.g., shorted distance or smallest required time. Decisions made by the navigation commonly have a long impact with a horizon of minutes up to hours (cf. [Sch11b]).

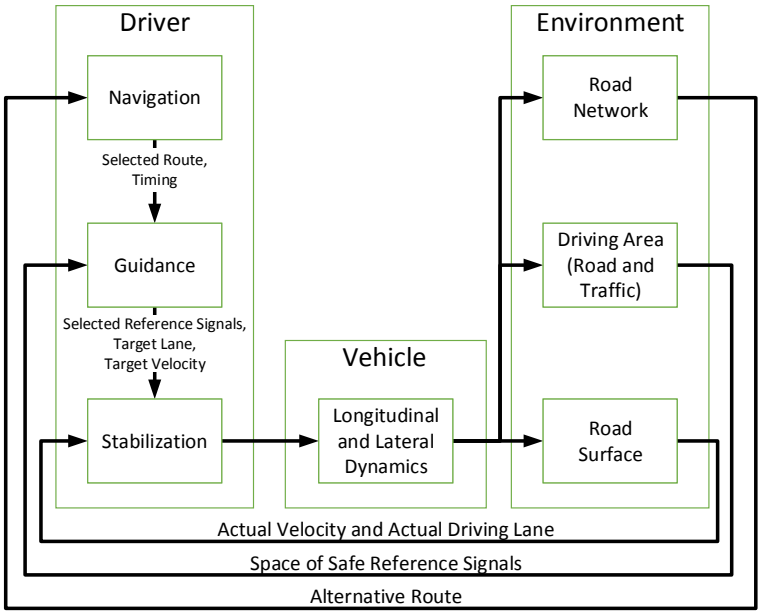


Figure 2.3.: Three layer model for the *driving task* according to E. Donges [Don82].

The dynamic process of driving road vehicle takes place on the two layers *guidance* and *stabilization* [Don15]. The movement in the static environment with other objects in motion results in a continuous change of the vehicle's environment and of the input information for the driver — especially the perspective representation of the three-dimensional world in the eyes of the driver (cf. [Don82]).

The second layer — *guidance* — addresses a time horizon of the next seconds up to minutes by deciding about reasonable and safe reference signals including the selected road lane and target speed (cf. [Sch11b]). At the same time, the driver should proactively intervene in order to create preconditions for minimal deviations of desired and actual reference signals (cf. [Don15]). For this task, the environment of the vehicle — the *driving area* — with the road, its lanes, and other vehicles are considered for generating the reference signals about the safe driving space. The safety of the own and other traffic participants on the road, e.g., other vehicles, is the crucial task of the guidance (cf. [Sch11b]).

These reference signals from the guidance are processed at the third layer — *stabilization* — for the immediate stabilization of the vehicle on the road. The driver has to align the vehicle motion with the reference signals by compensating emerging deviations in a closed-loop control process of corrective motoric actions (cf. [Don82]). The stabilization of the vehicle on different road surfaces is commonly supported by ADAS, e.g., ABS and ESC. The time horizon of stability tasks is instantaneous and does only considerably less than a few seconds. All decisions of the driver or ADAS are influenced by the *longitudinal* and *lateral dynamics* of vehicles. The impact actions initiated by the stabilization may differ on the real road from the anticipated impact in the processing of the stabilization.

SAE level	Name	Narrative Definition	Execution of Steering and Acceleration/ Deceleration	*Monitoring* of Driving Environment	Fallback Performance of *Dynamic Driving Task*	System Capability *(Driving Modes)*
***Human driver* monitors the driving environment**						
0	No Automation	the full-time performance by the *human driver* of all aspects of the *dynamic driving task*, even when enhanced by warning or intervention systems	Human driver	Human driver	Human driver	n/a
1	Driver Assistance	the *driving mode*-specific execution by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	Human driver and system	Human driver	Human driver	Some driving modes
2	Partial Automation	the *driving mode*-specific execution by one or more driver assistance systems of both steering and acceleration/ deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	**System**	Human driver	Human driver	Some driving modes
***Automated driving system* ("system") monitors the driving environment**						
3	Conditional Automation	the *driving mode*-specific performance by an *automated driving system* of all aspects of the dynamic driving task with the expectation that the *human driver* will respond appropriately to a *request to intervene*	System	**System**	Human driver	Some driving modes
4	High Automation	the *driving mode*-specific performance by an automated driving system of all aspects of the *dynamic driving task*, even if a *human driver* does not respond appropriately to a *request to intervene*	System	System	**System**	Some driving modes
5	Full Automation	the full-time performance by an *automated driving system* of all aspects of the *dynamic driving task* under all roadway and environmental conditions that can be managed by a *human driver*	System	System	System	**All driving modes**

Figure 2.4.: Levels of driving automation by the SAE [SAE14].

Feedback about internal parameters of the vehicle, the real position of the vehicle on the road, and changes of the proximate vehicle environment have to be gathered for repeated adjustment by the stabilization (cf. Fig. 2.3).

2.1.4.2. Levels of Automation for Road Vehicles

As shown in Fig. 2.4, the SAE defines six categories for the automation of vehicles on public roads (cf. [SAE14; SAE18]). The authors reside with the initial version SAE J3016-2014 ([SAE14]) from 2014 because the version SAE J3016-2018 ([SAE18]) has become more technical. The basic definition of the automation level has not changed. The reader is referred to [SAE18] for further information about the current revision of these automation levels. Each automation level is explained in the following:

***Level 0* - No Automation:** At automation level 0, no automation function is present in the vehicle. The driver is responsible for the monitoring of the vehicle's environment and the control of the longitudinal and lateral vehicle movement for the complete driving duration. The driver is also the fallback for the driving task in the presence of failures or critical situations. The driver has to react to emerging danger and return the vehicle into a safe driving state.

***Level 1* - Driver Assistance:** At level 1 — driver assistance —, systems support the driver in the control of either the longitudinal or lateral vehicle motion. The driver is required to supervise the operation of the assistance systems permanently and acts as a fallback in the presence of failures and critical situations.

Example for driver assistance systems of level 1 are ACC and LKAS (cf. [BR15]).

***Level 2* - Partial Automation:** At level 2 — partial automation —, systems take control of the longitudinal and lateral vehicle motion for given time durations in specific situations. The driver is still required to monitor the behavior of the vehicle and changes in its environment. At any time, the driver has to be prepared to take back the control of the vehicle as safety fallback in the presence of system faults or unsafe situations.

Examples of autonomous vehicle systems with level 2 automation are traffic jam assistants and highway assistants (cf. [BR15; Gas+12; Gas12]).

***Level 3* - Conditional Automation:** Systems with conditional automation take over the longitudinal and lateral control of the vehicle for given durations in specific situations. Opposed to level 2 — partial automation — , the human driver is *not* required to monitor the behavior of the vehicle and its environment continuously. The driver is allowed to occupy himself with other tasks. However, he remains the fallback for the driving task if the system operates at resp. outside its intended functionality and is unable to mitigate resulting risks. The driver can still intervene and take over the vehicle control at any time.

The handover of the vehicle control from the system to the human driver in critical situations has to be initiated by the system. The system has to identify its operation at resp. outside its intended functionality and provide the human driver with a sufficient duration for taking over the vehicle control. Throughout the handover period, the system has to maintain its safety and remain operational. The duration of this handover period is still subject to discussion and not yet generally defined by any standard.

In case the human driver does not take over the vehicle control in time, the system must perform appropriate safety maneuvers, e.g., an emergency stop, in order to transfer the vehicle into a safe state. In the *safe state*, any risks for the safety of passengers, other objects, and other persons have to be excluded. The identification and selection of safe states for autonomous vehicle systems in specific situations are currently subject to research (cf. [RM15]).

Definition 2.8 (Safe State)**.** *A safe state is defined as the state of the vehicle in which any risks for the safety of vehicle's passengers, other objects, and persons is excluded.*

An example of a system with conditional automation is a *highway chauffeur* (cf. [BR15]). A similar improvement of the automation can be achieved for the traffic jam assistant (cf. [Sch17]).

***Level 4* - High Automation:** At level 4 — high automation — systems take over the longitudinal and lateral control of the vehicle motion from the human drivers for the complete operation in given driving modes, e.g., driving of highways. Human drivers are not required to monitor the systems and their environments continuously or to take over the vehicle control in the presence of unsafe situations or system faults. Human drivers will have to act within a sufficient reaction time if the driving mode ends in foreseeable future. In comparison to level 3 — conditional automation — , autonomous vehicle systems with level 4 — high automation — act as fallback solutions. These highly automated systems are responsible for maintaining the safety of the vehicles in critical situations and the presence of system faults instead of the human driver. They must be aware of all limits for their intended functionality and be able to identify any violations of these limits in order to transfer the vehicle into a *safe state* without human intervention.

An example of a system with high automation is a highway pilot (cf. [BR15]).

***Level 5* - Full Automation:** The highest level of automation for vehicle systems is full automation. Systems of level 4 — high automation — can handle specific driving modes, e.g., driving on highways. However, systems with full automation must be able to operate in all possible driving modes, e.g., driving on rural and urban roads in addition to highways. These systems have to handle all situations — even in the presence of critical situations and system faults — because a human driver is not available as safety fallback. The operation of full automated vehicles must be able without any driver (cf. [Sch17]).

Robot taxis are an example of vehicles with full automation (cf. [BR15]).

Other organizations, like Federal Highway Research Institute (Bundesanstalt für Straßenwesen) (BASt), NHTSA, and German Association of the Automotive Industry (VDA), have defined their own taxonomies for the automation of vehicle systems (cf. [Gas+12; Gas12; US 13]). All these taxonomies can be aligned with the classification by the SAE and are not further discussed here. The reader is referred to [BR15] for additional information.
With an increasing level of automation, the driver loses responsibility for the vehicle control and vehicle safety while the system gains responsibility for the vehicle control and vehicle safety. For automation level 0 — no automation — to level 2 — partial automation —, the driver remains responsible for the safety of the vehicle in public traffic and has to continuously monitor the vehicle and its environment in order to interfere with the vehicle control if required. Beginning with level 3 — conditional automation —, the system substitutes the driver in the task of monitoring of the environment and with level 4 — high automation — as the safety fallback in the presence of critical situations

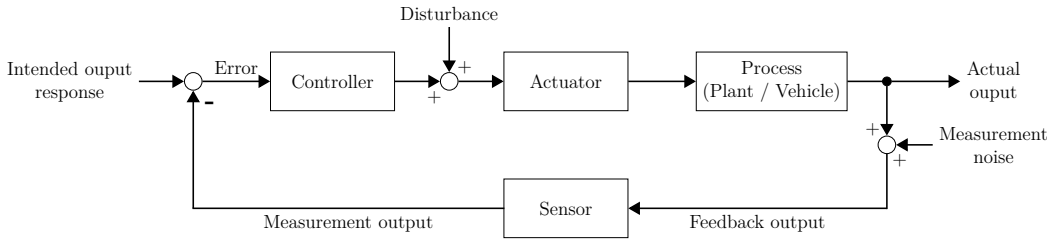
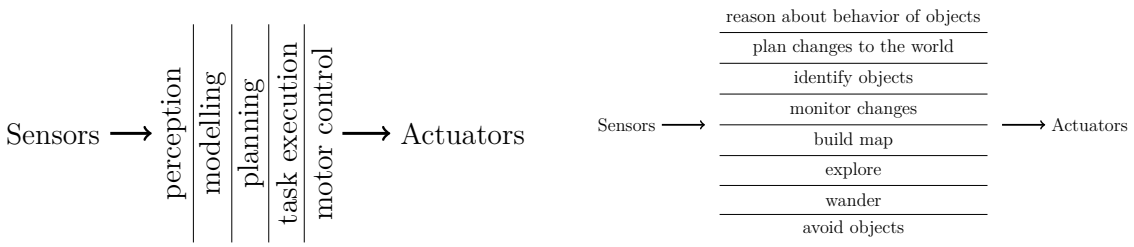


Figure 2.5.: Closed-loop feedback control system [DB10].

and system faults. This is especially important for the discourse about liability questions for collisions involving automated vehicles.
The general public does not recognize the large gap between today's available ADAS and full automated vehicles [NL14]. Today, systems with level 2 — partial automation — are predominantly deployed in production vehicles and first systems with of level 3 — conditional automation — have been recently introduced to vehicles of the luxury segment, e.g., the Audi AI traffic jam pilot in the Audi Q7 in 2017 (cf. [HBS18]). However, terms *autonomous vehicle* of *self-driving car* are commonly used by the general public when spoken about current and future automation of vehicles, but these terms refer to automation levels 4 — high automation — and level 5 — full automation — by the SAE. Design and implementation of autonomous vehicle systems have to be aligned with the definition of the *driving task* and the level of automation. The following section describes an underlying architecture for autonomous vehicle systems.

2.1.5. Functional Architecture of Autonomous Vehicle Systems

Autonomous vehicle systems are required to continuously monitor their environments via sensors in order to react to changes and deviations in their environments in accordance with its navigation goal. The real world continuously changes because its dynamic objects, e.g., other vehicles or pedestrians, behave autonomously. These objects may only react to decisions and actions of the automated vehicle but cannot be controlled by autonomous vehicle systems. The number of dynamic objects in the real world and the complexity of their behavior makes it unlikely for the autonomous vehicle systems to correctly anticipated all movements and actions for all dynamic objects in their vicinities.
Autonomous vehicle systems are generally developed as component-based systems with decision makers and closed-loop control systems (cf. [Den+14]). The decision makers can be viewed as replacements for the human operators. A *closed-loop control system* continuously corrects the output of a process via a feedback loop of relevant information (cf. Fig. 2.5) (cf. [DB10]). A *controller* compares the sensors measurements about the *process* from the *sensors* to the intended behavior of the *process* and corrects emerging deviations via *actuators* on the *process*.
The autonomous vehicle system encapsulates the *controller* as well as the decision maker while the vehicles and the real world represent the *process*. Vehicle sensors continuously perceive the environment of the vehicles — the real world — and the autonomous vehicle



(a) Decomposition into functional modules. (b) Decomposition into behaviors.

Figure 2.6.: Decomposition for the control system of autonomous robots [Bro86].

systems compare the sensor measurements with its internal environment representations. Based on changes of the environment, autonomous vehicle systems process adequate decisions on *navigation* and *guidance* level which are implemented as actions by the components of *stabilization* as input for the vehicle actuators (cf. Fig. 2.4). The actuators change the state of the vehicle and, therefore, the state of the vehicle's environment. The *control loop* via the vehicle sensors forwards all these changes back to the autonomous vehicle systems for additional correction to the vehicle behavior.

The data about the environment from the vehicle sensors are subject to measurement noise leading to deviations between parameter values in the real world and the measured parameters values, e.g., the object positions (cf. Fig. 2.5). Actuators, e.g., engine or brakes, may introduce further deviations into the control loop because they might be mechanically incapable of precisely implementing the actions from the system's *stabilization*. Imprecise sensor measurements and poor executions by actuators may impact the real behavior of vehicles in the real world.

The architectures of autonomous vehicle systems have to be aligned with the systems' level of automation and the various levels of the driving task (cf. Section 2.1.4). Autonomous vehicle systems can be decomposed in vertical and horizontal dimensions; alongside different system functions for the driving task (*vertical*) or alongside function components (*horizontal*) (cf. [Bro86]). The *horizontal* decomposition based on the information flow within the system into a series of functional units is a common practice for control systems (cf. Fig. 2.6a). These systems commonly consist of functional units for *sensing*, *modeling* sensor data in a world representation, *planning*, *task execution*, and *motor control* [Bro86].

In *vertical* dimension, autonomous vehicle systems can be decomposed alongside different classes of desired system functions for the driving task (cf. Fig. 2.6b). These classes are called *levels of competence* by Brooks (cf. [Bro86]). Each level of competence corresponds to a layer in the layered architecture (cf. Fig. 2.6b). At level 0, behavior for the avoidance of other objects is considered. More sophisticated *levels of competence*, e.g., reasoning about changes in the environment, are build as additional layers on top of lower, existing layers.

A complex system architecture, which considers vertical and horizontal decomposition, has been proposed by Matthaei in [Mat15]. This architecture is considered as reference in this thesis. Matthaei incorporates the localization of the vehicle and the three levels

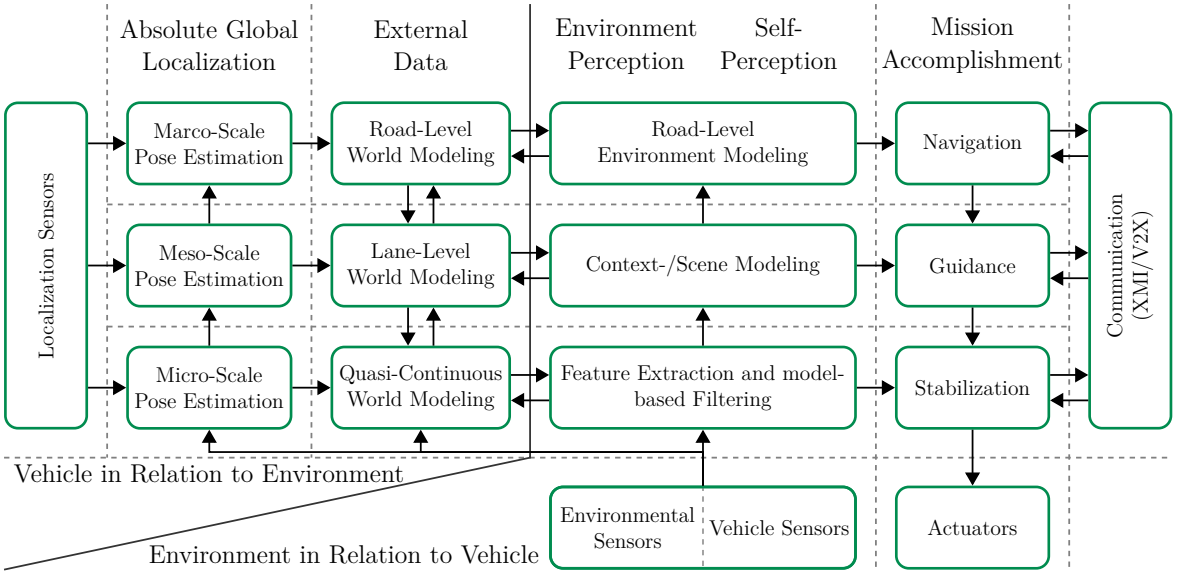


Figure 2.7.: Logical system architecture by Matthaei [Mat15].

of resolution — *macro-scale*, *meso-scale*, and *micro-scale* — proposed by Du et al. in [DMB04]. Fig. 2.7). The three levels correspond to the three level for the *driving task* by Donges (cf.[Don82]). Each level differs in its resolution, time and space accuracy, horizon, considered features, and cycle times (cf. [Mat15]):

Strategical (Navigation) level: The planning of routes at a macro-scale resolution on the road network for the duration of the trip.

Tactical (Guidance) level: The decision making about specific vehicle maneuvers, e.g., passing or lane changing, at a meso-scale resolution in the setting of lanes and vehicles for the next seconds to minutes.

Operational (Stabilization) level: The (reactive) stabilization of the vehicle as well as the execution of maneuvers at a micro-scale resolution for a time horizon of a few seconds and less.

The core of the functional system architecture — *absolute global localization*, *external data*, *environmental* and *self-perception* ,(perception), and *mission accomplishment* (cf. Fig. 2.7) — are surrounded by interfaces of the vehicle to the environment and driver — *sensors*, *actuators*, and *communication* equipment.[Mat15]
The columns *perception* and *mission accomplishment* are part of the *vehicle-referenced view* and have been addressed by previous publications (cf. [BD08; Dic07; Leo+08; Mon+08]). The *vehicle-referenced view* describes the environment in relation to the vehicle where a absolute global localization is not necessary. A *environment-referenced view* of vehicle and environment is described by the columns *absolute global localization* and *external data*. The environment is referenced in an *absolute global reference frame* while the pose of the vehicle is determined in relation to the environment.[Mat15]

The columns of the functional architecture are described in more detail in the following sections.

2.1.5.1. Absolute Global Localization

Absolute global localization is required for the processing of external data from other traffic participants via vehicle-to-vehicle (V2V) or vehicle-to-infrastructure (V2I) and the stabilization of the vehicle in environments without any environmental features, e.g., deserts. Data is commonly received from global navigation satellite systems (GNSSs) with varying level of accuracy. The data accuracy of data determines the usage of this data throughout the levels — strategic, tactical, and operational — of autonomous vehicle systems. Ordinary GNSSs receivers may have positioning errors up to 20m. Data from these inaccurate receivers can only be incorporated on the strategical level (macro-scale) for determining the macro-scale pose of the vehicle. For an application of this data on tactical (meso-scale) or operation level (micro-scale), additional methods, e.g., differential global positioning system (DGPS), for the improvement of the accuracy have to be incorporated. GNSS are currently not able to guarantee a sufficient accuracy for the micro-scale pose estimation future autonomous vehicle systems towards autonomous driving. Therefore, a local reference frame without a direct GNSS support is required for tasks of the stabilization.[Mat15]

2.1.5.2. External Data

External data encompasses all environmental data that has been perceived or generated outside the vehicle and consists of (cf. Fig. 2.7).

- the global vehicle pose, which is required for vehicle-to-X (V2X) and map data,
- the stationary environment — the scenery (cf. [Gey+14]) —,
- information about the dynamic environment, e.g., vehicles, traffic congestion, and hazards, and
- information about the state of other traffic participants via V2X.

The accuracy, details and update times of the *external data* is determined by the abstraction level of the architecture (cf. Fig. 2.7). For example, information about traffic congestions is commonly part of the strategical and tactical level of the architecture depending on the data's level of detail. Information about traffic lights and their current state require a more accurate level of detail in the model in order to associate the lights to their corresponding lanes. Therefore, traffic lights are part of the tactical level model. At the operational level, the model includes, e.g., the object lists of hypotheses about other traffic participants perceived via V2I (cf.[HW11]). [Mat15]
The modules of the column *external data* (cf. Fig. 2.7) focus on two task; the preparation of map data for the vehicle internal reference system and the fusion of data from multiple external sources into one cohesive model. The vehicle reference system requires to

determine a map relative pose for the processing by the modules of the *environmental perception* which can be obtained by correspondent map-matching approaches (cf. [QON07]).
For the usage of data from multiple external sources in the autonomous vehicle systems, environmental features from the different sources have to be correlated and aggregated into one cohesive model. At strategical level, an abstract course of the road with its topological information is stored in a history. For the tactical level, the history may include the course of each lane with associated speed limits or driving directions. The features addressed during operation level may be point landmarks (in an object-based representation) or parts of a grid-based representation. [Mat15]
In case an absolute global position is not part of the ADAS, an exchange of local data between traffic participants is not possible. The missing of a common reference frame from the GNSSs prohibits to align the world representations of different traffic participants. Without the exchange of data between traffic participants, each vehicle has to navigate in its own perceived world. [Mat15]

2.1.5.3. Environment-Perception and Self-Perception

The *perception*, covering the *environmental* and *self-perception* (cf. Fig. 2.7), aggregates and prepares all data from internal sensors and external sources about the vehicle's state and its surroundings for the later processing by the *mission accomplishment*. ADAS are only able to safely operate and avoid hazardous decisions with precious knowledge about the vehicle's state. The own state encompasses information about, e.g., the states of sensor and actuators, steering angle, wheel rotation rates, fuel, and battery states. Data processed by the *perception* is not only transferred to the *mission accomplishment* but also provided to external receivers via V2X in order to enable cooperation and collaboration among traffic participants. [Mat15]
Analog to the representation of environmental features by the *external data*, the representations of the vehicle's environment within the *perception* is hierarchically organized by the three abstraction levels — strategic, tactical, and operational — (cf. Fig. 2.7). At *operational level* precise and quasi-continuous values are extracted from sensor data. The *environmental perception* determines the features of objects, e.g., positions, postures, and motions, in the surrounding of the vehicle while the *self-perception* assembles a representation of the inner vehicle state. Features about, e.g., the vehicle state, weather conditions, traffic lights, lane markings, the position, and movement of other traffic participants (micro-scale representation) are independently processed by, e.g., object and lane tracking or grid-based algorithms. The processed features are transferred from the operational level to the higher tactic level and the *mission accomplishment* for a high-frequent closed-loop. [Mat15]
The *tactical level* addresses the context resp. scene modeling. The independent environmental features from the *operational level* are assembled into one associative context encompassing semantic information besides geometric and topological information. The *scene* (cf. Definition 2.25) combines features of the stationary environment — *scenery* (cf. Definition 2.27) — with features of dynamic objects (cf. Definition 2.29), e.g., vehicles in

the environment of the vehicle (meso-scale representation). For the scenery, objects, e.g., traffic lights and other static objects (cf. Definition 2.28), are associated to road lanes. [Mat15]
At *strategical level*, the environment of the vehicle is described for the route planning by a macro-scale level representation of the road topology. The topology consists of the high-level road network, and macroscopic traffic flows with the geometric and semantic information still being present. [Mat15]

2.1.5.4. Mission Accomplishment

The processing of the *mission accomplishment* is partitioned following the same hierarchy as the other columns (cf. Fig. 2.7):

- Navigation at *strategical level*, based on the road network and traffic flow processed by the *perception*,
- guidance and decision making at *tactical level* based on the abstracted local scene, which consists of features from the scenery and dynamic objects, and
- vehicle stabilization and execution of decisions at *operational level* based on exact geometric values of dynamic and static objects.

The scale in the processing of the overall mission, e.g., the route to the target destination, become more detailed with each lower level of abstraction (cf. Fig. 2.7). The stabilization during operation level incorporate more detailed information about the environment from the *perception* as the navigation at the operational level. The result of the processing by the *mission accomplishment* is the adjustment of control values for the vehicle actuators. [Mat15]
Following the route planning at *strategical level*, the next navigation point is forwarded from the *navigation* at the strategic level to the vehicle *guidance* at the tactical level. At *tactical level*, the abstract and application independent *scene* provided by the *perception* is assessed for the next navigation point and application relevant features are extracted based on mission-specific and permanent goals. The extracted features define the internal *situation* which is analyzed to determine an appropriate driving maneuver. A *situation* in the internal processing of autonomous vehicle systems is the *subjective* representation of the entirety of circumstances, conditions, options and determinants from an element's point of view for the selection of an appropriate behavior pattern at a particular point of time (cf. Definition 2.9). A detailed explanation and distinction of the terms *scene* and *situation* can be found in [Ulb+15]. The selection of the driving maneuver is performed by the guidance concerning traffic regulations and forwarded to the stabilization of the operation level. [Mat15]

Definition 2.9 (Situation)**.** *A* situation *in the internal processing of autonomous vehicle systems is the* subjective *representation of the entirety of circumstances, conditions, options and determinants from an element's point of view for the selection of an appropriate behavior pattern at a particular point of time (cf. [Ulb+15]).*

The selected driving maneuver by the guidance at the tactical level is executed at *operational level* by the vehicle stabilization. The operational level is characterized by its closed-loop control of the vehicle based on direct feedback from internal and external sensors. A trajectory with time- and space-based nominal positions for the vehicle is calculated based on the extracted features from sensor data of the *environmental perception*. A closed-loop controller calculates and manipulates the set values for the actuators, e.g., engine, brakes, and steering, in order to align the real world position of the vehicle with its nominal positions of the trajectory. [Mat15]
As shown in Fig. 2.7), results from the *mission accomplishment* are communicated internally to the passengers resp. the driver and externally to other traffic participants via

- the human machine interface (HMI),
- acoustic signals, e.g., horn and warning signals,
- optical signals, e.g., headlight, hazard lights, and turn indicators at the tactical level or brake lights at the operational level, and
- V2V communication channels.

External communication is especially important in the presence of critical environmental situations. An example of internal communication is the visualization of the processed route to the target destination.
Passengers may instruct autonomous vehicle systems solely on the strategical level by defining the target destination for their journey. For ADAS, additional possibilities of interaction on lower levels may exist. For example, a driver may interact with an ACC on the tactical level by adjusting the desired time gap to preceding vehicles. [Mat15]
The general hardware architecture of autonomous vehicle systems is investigated in the analysis of the development process for autonomous vehicle systems in Section 3.3.2. The following section gives an introduction to *simulation-based testing* as an important verification technique for autonomous vehicle systems. Comprehensive verification and validation of autonomous vehicle systems are essential for their deployment and operation in production vehicles on public roads.

2.2. Simulation-based Testing of Autonomous Vehicle Systems

According to the Institute of Electrical and Electronics Engineers (IEEE) [CS217], *testing* is one of three traditional verification techniques and aggregates a wide field of diverse but incomplete methods for finding bugs and errors in programs [Leu11; LS09]. Myers defines *testing* as "the process of executing a program with the intent of finding errors". Testing as well as verification involves a system or model of the system $\mathcal{M}$, a set of (system) parameters P, a set of (system) inputs U, and a property ψ, which shall hold

for the system. Testing and verification activities are then defined in terms of system behavior. The system behavior of system $\mathcal{M}$ is denoted as $\Phi(\mathcal{M}, p, u)$ with $p \in P$, $u \in U$, and u is a function of time. $\Phi(\mathcal{M}, P, I)$ defines all possible behavior of system $\mathcal{M}$ under parameters in P and inputs in U. *Testing* is then formally defined as $\Phi\left(\mathcal{M}, \hat{P}, \hat{U}\right) \models \psi$ given finite set $\hat{P} \in P$ of parameters, given finite set $\hat{U} \in U$ of inputs and a given property ψ which shall hold for the system. The sets $\hat{P}$ and $\hat{U}$ are finite. The finite set of (test) input $\hat{U}$ in the tests results finite space of tested (system) parameters $\hat{P}$ for the system $\mathcal{M}$. [Kap+16]

Definition 2.10 (Testing)**.** *"Testing is the process of executing a program with the intent of finding errors." [MSB11]*

Errors are commonly defined in the context of *faults*, *failures*, and system *malfunctions* because a strong relationship exists between them. However,definitions for these terms slightly vary throughout academic and industrial publications.
An *error* is defined in the safety standard ISO 26262 as "discrepancy between a computed, observed or measured value or condition and the true, specified, or theoretically correct value or condition" [Int09a] and by Liggesmeyer as a mistake by an engineer in the (software) implementation of a system [Lig09]. Under specific runtime condition, *errors* result in *faults* at execution.

Definition 2.11 (Error)**.** *"A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition" [Int09a].*

A *fault* is defined in safety standard ISO 26262 as an "abnormal condition that can cause an element or an item to fail" [Int09a] and by Isermann as "unpermitted deviation of at least one characteristic property (feature) of the system from the acceptable, usual, standard condition."[Ise06b]. A fault represents an abnormal state within the system. A fault can be *dormant* or *active*. While a dormant fault has no impact, an active fault impacts the externally observable system functionality. Another term for fault is *defect* (cf. [Lig09]).

Definition 2.12 (Fault)**.** *"An abnormal condition that can cause an element or an item to fail" [Int09a].*

An active *fault* results in a (system) *failure*. *Failure* is defined in safety standard ISO 26262 as "termination of the ability of an element or an item to perform a function as required"[Int09a] and by Isermann as "a permanent interruption of a system's ability to perform a required function under specified operating conditions" [Ise06b].

Definition 2.13 (Failure)**.** *"The termination of the ability of an element or an item to perform a function as required" [Int09a].*

Isermann further defines the term *malfunction* as "an intermittent irregularity in the fulfillment of a system's desired function" [Ise06b]. The safety standard ISO 26262 defines a *malfunctioning behavior* as "*failure* or unintended behavior of the item with respect to the design intent for this item".

Definition 2.14 (Malfunction)**.** *"An intermittent irregularity in the fulfillment of a system's desired function" [Ise06b].*

A comprehensive taxonomy for *failures* and *faults* is given in [Avi+04]. However, Avizienis et al. use a slightly divergent definition for *error*, *fault*, and *failure*. This thesis follows the definition of the safety standard ISO 26262 for these terms.
System engineers use testing activities and other verification techniques to increase their confidence that systems meet their requirements as well as relevant safety standards, e.g., the safety standard ISO 26262. Thus, testing and verification account for a significant portion of effort in the development of autonomous vehicle systems. *Simulation-based testing* approaches offer improved means of testing and verification for autonomous vehicle systems with a significant benefit for the overall development costs. *Simulation-based testing* constitutes the testing of systems using models and simulations (cf. Definition 2.15).

Definition 2.15 (Simulation-based Testing)**.** *Simulation-based testing is testing of systems using models and simulations.*

The substantial difference between regular testing and simulation-based testing is the system under test (SUT) (cf. Definition 2.16), whose behavior is analyzed and verified. Testing executes the real systems and verifies actual instances of system behavior while simulations predominantly verify models and their inherent approximations of the system behavior (cf. [Kap+16]).

Definition 2.16 (System under Test)**.** *The target of testing and simulation is referred to as system under test (SUT). Other terms used for the system under test (SUT) are* artifact under test *or* test object*.[KKL13]*

Simulations are applied in a variety of fields: manufacturing, construction engineering, project Management, logistics, transportation, business processes, or health care [Ban+05]. Shannon defines *simulation* as "the process of design a *model* [(cf. Definition 2.19)] of a real *system* [(cf. Definition 2.1)] and conducting experiments with this model for the purpose of understanding the behavior of the system and/or evaluating various strategies for the operation of the system". The Association of German Engineers (VDI) defines *simulation* as the implementation of a dynamic *systems* in a *model* suitable for experiments to gain knowledge that can be transferred (back) to the reality (cf. [VDI14]). Kapinski et al. formally define *simulation* analogous to *testing* and *verification* as the process of obtaining a numerical estimation of system behavior $\Phi\left(\mathcal{M}, \hat{P}, \hat{U}\right)$ for a specific collection of operation conditions given by given finite set of parameters $\hat{P} \in P$ and finite set of inputs $\hat{U} \in U$.

Definition 2.17 (Simulation)**.** *The process of designing a* model *of a real system and conducting experiments with this* model *to understand the behavior of the* system *and evaluating various strategies for the operation of the* system.*[Sha98]*

Simulations offer the possibility to analyze the existing system as well as predict the performance of novel systems under development. As an analysis tool, simulations enable to simulate changes of existing systems and analyze the impact on their system behavior. As a design tool, simulations allow to simulated models of novel systems, which are in the preliminary or planning stage of their development, and predict their performance under diverse sets of conditions [Ban+05; Sha98].
Simulation has to be distinguished from *emulation*. Simulation describes a *system* as a mathematical model, while *emulation* mimics the *system* and can be used as substitution of the original system (cf. [Sch05]). The concrete definition by Schmitt is given in Definition 2.18.

Definition 2.18 (Emulation)**.** *The replication of system hardware (components) and its (their) behavior by other hardware (components) (cf. [Sch05]).*

The scientific analysis of systems often requires sets of assumptions about their functionality and behavior over time. These assumptions are expressed in *models* by mathematical, logical, and symbolic relationships between entities and objects of the systems. These *models* can be used in the system analyses for the representation of the system functionality (cf. [Ban+05; LK97]). Shannon defines a *model* as "a representation of a group of objects or ideas in some form other than that of the entity itself"[Sha98]. The Association of German Engineers (VDI) defines *models* as simplified representations of a planned or existing real system in another conceptual or representational system (cf. [VDI14]).

Definition 2.19 (Model)**.** *"A representation of a group of objects or ideas in some form other than that of the entity itself".[Sha98]*

Most analyses require to solely consider aspects of real systems for their models which are relevant to the problem under investigation in the analysis (cf. [Ban+05; VDI14]). Therefore, models are often simplifications of real systems but have to be sufficiently detailed in order to permit valid conclusions about the real system (cf. [Ban+05]). The definition of appropriate system models $\mathcal{M}$ is generally named *modeling* [Kap+16]. Different analyses might require different models of the same real systems (cf. [Ban+05]). Many people associate the term *model* with *physical models*, e.g., clay cars in wind tunnels, but those models are usually not the type of models which are considered in system analyses of autonomous vehicle systems (cf. [LK97]). The second class of models is *mathematical models*. Mathematical models represent the real system by symbolic notations and mathematical equations that are manipulated in order to analyze the reaction of these models and, therefore, to deduce the behavior of the real systems. This deduction requires the mathematical model to sufficiently model relevant aspects of the

real systems (cf. [LK97]). *Simulation models* used in simulations are a particular type of mathematical model [Ban+05]
In case mathematical models are sufficiently simple, it is possible to solve these models and get an exact *analytical solution mathematically* by using, e.g., differential calculus, probability theory, algebraic methods, or other mathematical techniques (cf. [LK97]). However, the complexity of many systems — especially systems operating in the real world — is so high, that valid mathematical models of these systems exhibit such high complexity, that an analytical solution of these complex models is virtually impossible. In this case, models may be studied in computer-based simulations by imitating the behavior of the systems over time. The models are "run" rather than solved; the inputs of the simulation model are numerically exercised in order to evaluate their impact on the model's outputs. An artificial history of the system is generated, and observations are collected in order to analyze and estimate the behavior of the real system.[Ban+05; LK97]
Simulation models may be classified with respect to different properties [Ban+05; LK97]:

static vs. dynamic: A static simulation model, sometimes called a Monte Carlo simulation (cf. [Moo97]), represents a system at a particular point in time. Dynamic simulation models represent systems as they change over time.

deterministic vs. stochastic: Deterministic simulation models contain no random variables and have known sets of inputs and outputs. A stochastic simulation model has at least one random variable as input. Random input variables lead to one or more random outputs for models. Randomized model outputs can only be considered as estimations for the characteristics of real systems.

discrete vs. continuous: Discrete and continuous models are defined analogous to discrete and continuous systems (cf. Definition 2.1). Variables of discrete models change instantaneously at specific discrete points in time, while state variables of continuous models change continuously over time.

Discrete simulation models are not purely used to model discrete systems, nor are continuous simulation models solely used to model continuous systems. In some simulations, even mixed-discrete-continuous simulation models may be used. The choice depends on the characteristics of the analyzed systems and the objects of the analysis.

Law et al. correspondingly distinguish between discrete (event) simulations and continuous simulations (cf. [LK97]).
For autonomous vehicle systems various elements of their control loops, like the vehicle's environment or the vehicle dynamics, have to modeled and simulated. The following section describes the various models in simulation-based tests of the autonomous vehicle systems.

2.2.1. Components of Simulation Frameworks

Testing of autonomous vehicle systems in simulations requires consideration of the complete control loop between the autonomous vehicle system, vehicle, and vehicle environment (cf. [Tel12]). Relevant aspects of systems and their environments, which impact the behavior of these autonomous vehicle systems, have to be modeled in the simulations. Formally, the autonomous vehicle systems and the models for relevant aspects of the control loop represent the model $\mathcal{M}$ (cf. Definition 2.10) [Kap+16].
A monolithic model for all relevant aspects is rarely defined. The complexity of a monolithic model would be very high and would not allow the usage of these models in x-in-the-loop (XIL) tests (cf. Section 2.2.2). Relevant aspects are individually modeled without the coupled problem in mind. Depending on the level of detail by the simulations, distinctive models are defined for relevant aspects, e.g., driver, vehicle dynamics, vehicle sensors, roads, and other traffic participants (cf. Fig. 2.8). Persistent interfaces allow defining models which precisely reproduce the corresponding entity with limited complexity independent from the development state of other models (cf. [HK16]). For example, the decisions of drivers can be modeled by *stochastic* models while the vehicle dynamics is modeled *deterministically* by, e.g., differential equations.
The simulation has to verify the behavior of the SUT over time and, therefore, requires all models to be *dynamic*. However, the cyclic data dependency between models themselves as well as these models and the SUT prohibit the usage of *continuous* models. For example, the model of the environment requires information from the model of the vehicle dynamics which requires input from the autonomous vehicle system, which controls the vehicle based on the current state of the environment model.
Models and the SUT are integrated into a *co-simulation* (cf. [Zel+10]). The control loop of the simulation is processed in discrete time steps (cf. [Neu14]). The *discrete* models are individually processed in a sequence of the control loop based on data from related models and the SUT. The output of the SUT is considered for the simulation of its environment in the current time step while the corresponding changes in the environment model are processed by the SUT in the next processing cycle. Small cycle times allow for nearly realistic and continuous simulations. The processing of models and the flow of data within the simulation is controlled by the *simulation framework* (cf. [NDW09]).

Definition 2.20 (Simulation Framework)**.** *A simulation framework (also called simulation environment) integrates an (autonomous vehicle) system under test (SUT) into a co-simulation with various models for the different aspects of the system's environment.*

In academic literature also the term *simulation environment* is used for simulation frameworks. This thesis primarily uses *simulation framework* in order to distinguish it from the real environment resp. the environment modeling and simulation within the simulation framework.
Simulations are distinguished between *open-loop* simulations and *closed-loop* simulations depending on the control loop in the simulations (cf.[Kap+16]):

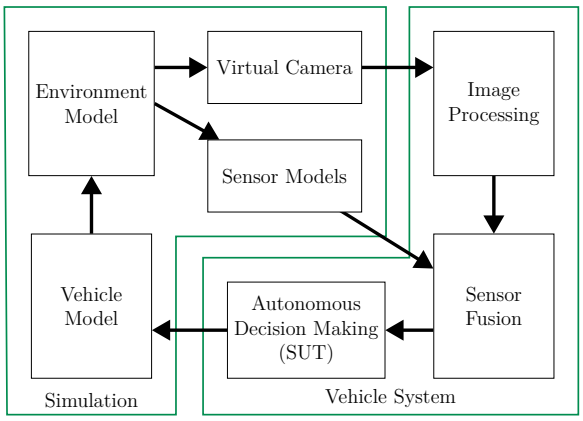


Figure 2.8.: Control-loop for simulations of autonomous vehicle systems [BR15].

Open-Loop Simulation: In an open-loop simulation, the output of the SUT are not considered for the simulation of the environment and inputs of the system under test. The behavior of dynamic objects in the environment is entirely independent of the behavior by the automated (ego) vehicle. The dynamic objects do not react to any maneuvers by the automated (ego) vehicle.

> **Definition 2.21** (Open-loop Simulation)**.** *Outputs from the system under test (SUT) are not considered for the simulation of the environment and inputs of the SUT. The feedback loop between the SUT and environment is not closed.*

Closed-loop simulation: In a closed-loop simulation, the feedback loop between the SUT and the simulation of the environment is closed. Outputs by the system are considered as input for the environment. Other (dynamic) objects in the environment consider the behavior of the SUT for the processing of their behavior. The system and the environment influence each other in either direction.

> **Definition 2.22** (Closed-Loop Simulation)**.** *Closed-loop simulations incorporate feedback loops to transfer the outputs of the system under test (SUT) via the environment simulation back to the system under test as inputs. Other (dynamic) objects in the environment react to the behavior of the SUT.*

While open-loop simulations are easier to establish, closed-loop simulations offer a greater realism because environment objects react to the behavior of the SUT.
Figure 2.8 (cf. [BR15]), depicts the components and data flow of a simulation framework for system tests of autonomous vehicle systems. The left side of Fig. 2.8 shows the virtual world with the virtual vehicle model for the longitudinal and lateral *vehicle dynamics*, the virtual environment for the simulation of the road infrastructure and traffic, and the sensor models for the perception of the virtual environment. The right side of Fig. 2.8

contains the components of the autonomous vehicle system for the processing of the sensor data and the decision making as they are later used in the vehicle. [BR15]
The virtual images from the camera model are analyzed by the *image processing* and integrated by the *sensor fusion* with the data from other *sensor models*, e.g., LIDAR and radio detection and ranging (RADAR) (cf. [RG05]), into a cohesive representation of the vehicle environment. This environment representation is used by the *(autonomous) decision making* — the SUT — to process the trajectory for the automated (ego) vehicle. The virtual *vehicle dynamics* process the input for the vehicle actuators, e.g., brakes, engine, and steering, from the *decision making*. The model of the vehicle dynamics changes the position and orientation of the automated (ego) vehicle in the virtual world. In addition to the behavior of automated (ego) vehicle, other traffic participants in the *environment model* move autonomously. Changes in the environment model influence the input of the sensor models in the following processing cycle. [BR15]
Simulation models have to sufficiently represent the real world for valid and reliable simulation results. Validation of these models is essential — especially for models of physical vehicle components, e.g., sensors and dynamics. For example, the simulation framework *PRESCAN* uses validated physical sensor models for RADAR, LIDAR, and camera vision [Gie+06]. Model validation is distinguished into two types (cf. [Nat98]):

External validation is the comparison of model output with (operationally relevant) observations from the real system. The addition of an uncertainty analysis helps to identify model inadequacies which are responsible for differences between model outputs and real observations. Uncertainty analysis examines the variability of the model's output in relation to the variability in the model's inputs, [Nat98]

Sensitivity analysis is an analysis of the relationship magnitude between inputs and outputs of models. A sensitivity analysis will indicate inadequacies in models if the direction and magnitude of sensitivity of model outputs to various inputs do not match the subject matter expertise.[Nat98]

The individual components and their models are described in the following sections in more detail.

2.2.1.1. Environment

The vehicle environment is essential for the testing of autonomous vehicle system in simulations (cf. Fig. 2.8). The environment model defines the virtual world as a representation of the real world and contains all relevant objects $\mathbf{O}$ for the simulation [Neu14]. It includes the road topologies, vehicles, and pedestrians as traffic, as well as the automated (ego) vehicle. The automated (ego) vehicle moves through the virtual world under consideration of the road infrastructure and other objects $\mathbf{O}$. Neumann-Cosel defines the *environment* to include all objects $o \in \mathbf{O}$ within a defined range $r_E \in \mathbb{R}$ around the automated (ego) vehicle (cf. Definition 2.24). This thesis considers the environment model to contain all (relevant) objects within the maximal range $r_E \in \mathbb{R}$ of the vehicle sensors.

Definition 2.23 (Object)**.** *An Object $o \in \mathbf{O}$, is an atomic entity of the environment or a combination of atomic entities. The atomicity of objects is subject to the level of detail in the simulation.[Neu14]*

Definition 2.24 (Environment)**.** *The (vehicle) environment* **Env** *is the set of all objects $o \in \mathbf{O}$, whose distances $d(E, o) \in \mathbb{R}$ to the automated (ego) vehicle E is smaller than the defined range $r_E \in \mathbb{R}$:* $\mathbf{Env}_E = \{o \in \mathbf{O} \mid d(E, o) \leq r_E\}$ *(cf. [Neu14]).*

The behavior of the automated (ego) vehicle and other traffic participants result in changes for the environment model at specific points of time over the duration of the simulation $\mathbf{T} : t_0, \ldots, t_i, \ldots, t_n$. *Discrete simulations* abstract from continuous changes by evaluating the static representations of the environment at these specific points in time t_i. These static representations are denoted as *scenes* (cf. Definition 2.25) and represent configurations of the vehicle's environment as an spatial-temporal arrangement from an observers point of view — including the *scenery*, dynamic objects, and self representation (cf. [Mau00a]). The *scene* preserves the state z of each object $o \in \mathbf{O}$ in the environment for a given point in time. The object state z_o encapsulates all relevant information about the object $o \in \mathbf{O}$ for simulation — including its position and orientation. Ulbrich et al. enrich the scene with information about the actors' and observers' self-representations (cf. [Ulb+15]).

Definition 2.25 (Scene)**.** *The configuration of the vehicle's environment as a spatial-temporal arrangement from an observers point of view — including the* scenery*, dynamic objects, and self-representation (cf. [Mau00a]).*

Definition 2.26 (Object State)**.** *The object state z_o encapsulates all relevant information about an object $o \in \mathbf{O}$, e.g., position and orientation.*

The *environment* can be separated into two parts: *static* and *dynamic environment*. The *static environment* is denoted as *scenery* and is presented in section Section 2.2.1.1.1. The *dynamic environment* is describe in terms of traffic in Section 2.2.1.1.2.

2.2.1.1.1. Scenery

The *scenery* encompasses all geo-spatially stationary aspects of the *scene* and, therefore, encapsulates all objects $o \in \mathbf{O}$ whose states z_o do not change over time. This entails information about roads, lanes, lane markings, road surfaces, or the roads' domain types as well as houses, fences, curbs, trees, traffic lights, or traffic signs (cf. [Ulb+15]). Sceneries are modeled once before corresponding simulations and remain consistent throughout these simulations.

Definition 2.27 (Scenery)**.** *The scenery S encapsulates the set of static objects $o \in \mathbf{O}$ in the environment $\mathbf{Env}_E$ (cf. [Neu14]).*

Definition 2.28 (Static Object)**.** *A static object $o \in \mathbf{O}$ does not change its state z_o throughout the complete simulation (cf. [Neu14]).*

In addition to the static content of the environment, traffic has to be considered for simulations as the environment's dynamic objects.

2.2.1.1.2. Traffic

Autonomous vehicle systems require the simulation of the road traffic as dynamic part of the environment. Dynamic objects, e.g., vehicles and pedestrians, have to perform specific maneuvers in order to stimulate reactions by the SUT. A *dynamic object* is an object $\hat{o} \in \mathbf{O}$ which changes its state $z_{\hat{o}}$ over the duration $\mathbf{T}$ of simulations.

Definition 2.29 (Dynamic Object)**.** *A dynamic object is an object $\hat{o} \in \mathbf{O}$ which can change its state $z_{\hat{o}}$ for each time stamp $t_i \in \mathbf{T}$ of simulations (cf. [Neu14])*

Dynamic objects are placed in the environment model in relation to objects of the scenery, e.g., roads and their lanes, with an initial state z_0. The logic for the behavior of dynamic objects defines changes of the object state $z_{\hat{o}}$ over the duration $\mathbf{T}$ of the simulation for each dynamic object $\hat{o} \in \mathbf{O}$. For example, maneuvers are defined in conjunction with velocities and accelerations for vehicles.
Traffic can be simulated with different levels of detail; as traffic flows with or without individual vehicles, as individuals objects, as individual objects with individual subcomponents. The reader is referred to [Sch17] for a detailed description of the different detail levels for traffic simulations.

2.2.1.2. Vehicle

In addition to the SUT, the remainder of the vehicle which is not part of the SUT has to be modeled. This may include vehicle sensors and vehicle dynamics (cf. Fig. 2.8). Models of vehicle sensors address the perception of the virtual environment and its virtual objects for input to the SUT. Sensor models are described in Section 2.2.1.2.1. The models of the vehicle dynamics define how the output of the SUT changes the state of the vehicle in the virtual world. Models of the vehicle dynamics are described in Section 2.2.1.2.2.

2.2.1.2.1. Vehicle Sensors

Autonomous vehicle systems require input from a perception of the virtual vehicle environment in simulations (cf. Fig. 2.8). Real sensors are substituted by sensor models which transfer the state of the environment at each time point $t_i \in \mathbf{T}$ into valid inputs,

e.g., object lists or points clouds, for the SUT (cf. [Tel12]). The input from sensor models will have to be further processed by additional environment perception algorithms if these algorithms are not part of the SUT (cf. image processing and senor fusion in Fig. 2.8). Sensor models may have varying levels of detail ranging from perfect *high-level* to realistic (*low-level*) sensor models. Idealized high-level sensor models provide the SUT with the exact information, e.g., position and orientation, of relevant objects in the virtual environment. Realistic low-level sensor models try to mimic real sensor hardware as closely as possible. This imitation commonly includes the imitation of the physical measurement principle of the real sensors in order to provide the SUT with realistic data about the environment — including data jitter. For example, realistic RADAR models may simulate the propagation of radio waves through the virtual environment. Realistic low-level sensor models help to evaluate and improve the robustness of autonomous vehicle systems to jittered input data. [Bel+12; Ste+15]

2.2.1.2.2. Vehicle Dynamics

While sensor models are concerned with the perception of the virtual environment in simulations, models of the vehicle dynamics address changes of the automated (ego) vehicle in the virtual world in response to the output of the SUT. The models of the vehicle dynamics have to mimic the dynamic movement of the vehicle in the real world depending on the level of realism in simulations. Specified models, e.g., the two-track model [Gie+06], have to sufficiently consider the inherent inertia of the vehicle and variances in forces by vehicle actuators, e.g., engine, brakes, and tires, for the longitudinal and lateral movement of the vehicle.

2.2.1.2.3. Driver

Instead of deterministic logic for the behavior of traffic participants (cf. Section 2.2.1.1.2), specific driver models can be introduced for vehicles in the virtual environment. These driver models commonly represent the behavior of real drivers stochastically where each possible driving maneuver is assigned with a possibility. In simulations, the vehicle maneuvers are nondeterministically chosen under consideration of their possibility.
Stochastic driver models have the disadvantage that they impede the validation of simulation results. Repetitions of simulation with stochastic driver models commonly result in different results because the behavior of vehicle deviates due to the nondeterministic decisions by these driver models.
Hochstaedter et al. separate the driver model into a decision model and an action model. The decision model calculates a trajectory as action intention under consideration of the environment, while the action model sets the setpoint values for the vehicle actuators, e.g., engines and braking system, in order to follow the trajectory. The action model interacts with the model of the vehicle dynamics for the correct execution of the setpoint values. [HZB00]
In the automotive domain, the individual components are predominantly used for XIL simulations. These tests are described in the following section in more detail.

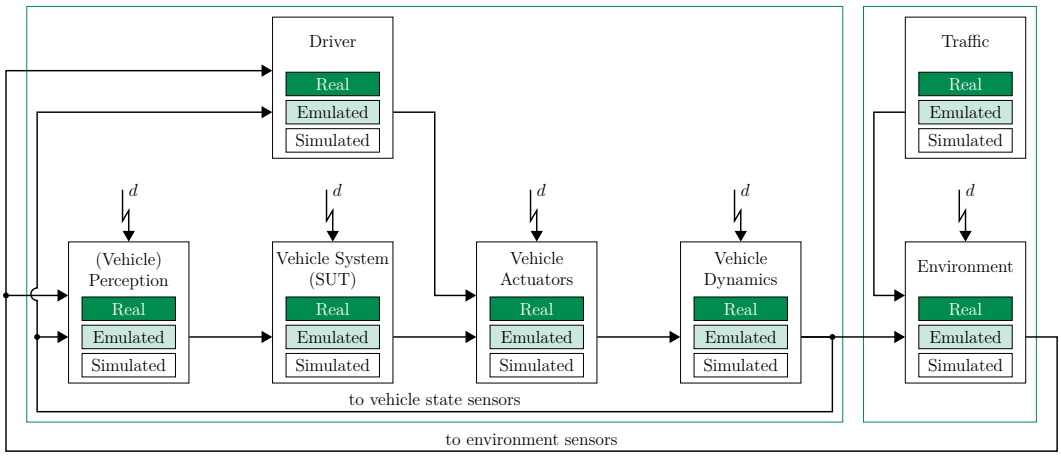


Figure 2.9.: Realization of control loop elements in XIL simulations (cf. [Gie+06]).

2.2.2. X-in-the-Loop Simulations

In the automotive domains, autonomous vehicle systems are predominately verified in x-in-the-loop (XIL) simulations because they allow for an efficient and cost-effective testing process (cf. [BNF16]). Autonomous vehicle systems can be systematically tested without disturbances from other connected but unrelated systems.
XIL simulations incorporate simulation models and real system components in varying degree in order to reproduce the control loop between SUT and their environments. As shown in Fig. 2.9 (cf. [Gie+06]), each element of the control loop, e.g., driver, vehicle system, vehicle dynamics, vehicle sensors, vehicle actuators, environment, and traffic, are either represented by a simulation model, emulated by hardware, or implemented by the real system component. Additional disturbances d can be injected for vehicle and environment elements of the control loop. This controlled injection of disturbances d allows verifying the dependability of the SUT (cf. [Gie+06]). The configuration of XIL simulations depends on the SUT and the target of the tests (cf. [Neu14]).
In the following sections, the different types of XIL simulations are described and classified in Kiviat diagrams (cf. [MR82]) based on the way elements of the control loop are integrated in the simulations. The dimensions of the Kiviat diagram corresponds to previously presented components of simulations (cf. Section 2.2.1): *vehicle system*, *driver*, *vehicle*, *dynamics*, *perception*, *traffic participants*, and *scenery* (cf. [Sch17; Str12]). The scale of all dimensions is three folded: (1) *simulated* by models, (2) *emulated* by hardware, and (3) as *real* system component. Some approaches are able to provide multiple solutions for the consideration of system components (dimensions). This is depict by solid areas in the Kiviat diagrams. In addition to the XIL approaches, real-world testing is identically classified for completeness.
The individual dimensions of the Kiviat diagrams are described in the following (cf. [Sch17]):

Vehicle system: The *vehicle system* represents the SUT and encompasses function components and the corresponding execution platform as the part of the vehicle,

which is verified in the XIL simulation. In case of autonomous vehicle systems, this can be a function on a real ECU, a function emulated by related hardware or a pure software function without any execution hardware.

Driver: The dimension *driver* addresses the behavior of the driver and its interaction with the vehicle via the HMI (cf. [Ina06]). The driver behavior can be subject to a real driver, emulated by a driving robot, or simulated by virtual driver models.

Vehicle: The dimension *vehicle* describes the remainder of the vehicle which is not part of the *vehicle system*. This may include vehicle components, e.g., the vehicle chassis. XIL approaches may incorporate real vehicles for realistic results. Otherwise, the vehicle can be emulated by similar vehicles or identical hardware. In software-in-the-loop (SIL) tests, the interaction of *vehicle system* and the remainder of the vehicle is commonly modeled by rest-bus simulations.

Dynamics: *Dynamics* describe the behavior of the vehicle on the road. In case of XIL approach incorporate a real vehicle, the real vehicle dynamics are present. Otherwise, a vehicle with similar dynamics can be used to emulate the dynamics of the original target vehicle. A simulated vehicle dynamics uses a model to process the motion of the vehicle in the virtual world. Thus, the vehicle behavior is not implemented in the real world.

Perception: The dimension *perception* addresses the remainder of the vehicle's perception system which is not part of *vehicle system*. A real perception will be present if environment perception is implemented on the later target platform. Otherwise, a similar hardware platform can be used to emulate the vehicle perception. Artificial generated object lists or sensor data as input to the *vehicle system* represent a simulated perception.

Traffic participants: The dimension *traffic participants* addresses the realization of other dynamic objects in the tests, e.g., vehicles, cyclists, and pedestrians. Traffic participants can be real if the tests are performed in the real world on public roads or testing grounds. The usage of crash targets or balloon vehicles represents an emulation of traffic participants. Traffic participants will be present in the simulation if they are modeled and implemented as software.

Scenery: The *scenery* describes the static environment of the *vehicle system*. Tests in the real world commonly incorporate the real scenery. An emulation of the scenery will be present if public roads are rebuilt at testing grounds by, e.g., artificial roads, lanes, curbs, and vegetation. In case, scenery objects are part of the virtual world which is generated by a simulation framework; a simulated scenery is present.

The various types of XIL simulations are described each in more detail in the following sections.

2.2.2.1. Model-in-the-Loop Simulations

Model-in-the-loop (MIL) simulations allow the evaluation of initial system designs early in the development process (cf. [BNF16]). A model of the system is integrated into a control loop with models of vehicle dynamics, sensor, actuators, environment, and traffic [GRS14; Ise06a]. As shown in Fig. 2.10, all dimensions of the simulation are modeled and simulated. The model of the system represents the model $\mathcal{M}$ in Definition 2.10.

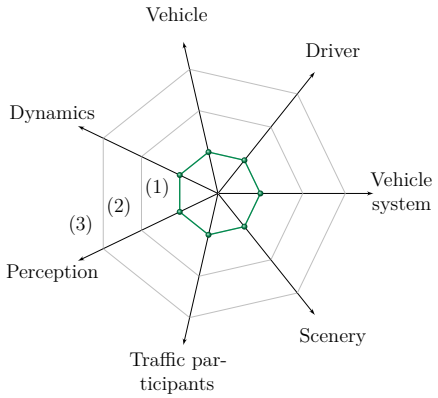


Figure 2.10.: Kiviat digram of MIL and SIL simulations [Sch17].

2.2.2.2. Software-in-the-Loop Simulations

Software-in-the-loop (SIL) simulations share extensive similarities to model-in-the-loop (MIL) simulations. The model of the as SUT is substituted by its real software implementation (cf. model $\mathcal{M}$ in Definition 2.10) [GRS14; Ise06a; Kap+16]. As for the MIL simulations all other entities of the control loop, e.g., vehicle dynamics, sensor, actuators, environment, and traffic, are modeled and simulated (cf. Fig. 2.10) [Sch17]. Novel software implementation can be evaluated in SIL simulations without any real hardware components.

2.2.2.3. Driver-in-the-loop simulations

Driver-in-the-loop (DIL) simulations focus on the behavior of the driver and his interaction with the vehicle and its systems. Therefore, the model of the driver is substituted by a real driver. Human drivers are seated in artificial vehicles on fixed-base (stationary) or motion-based simulators with projections of the virtual environment and vehicle-like controls (cf. [Sch17; Ste+15]).
Most dimensions of driver-in-the-loop (DIL) simulations are simulated — except for the remainder of the vehicle and the vehicle dynamics (cf. 2.11):

- Vehicle dynamics are simulated in fixed-base simulators while a motion-based simulator emulates the vehicle dynamics by a motion platform. This platform reenacts the forces on the vehicle based on the simulation of the environment.

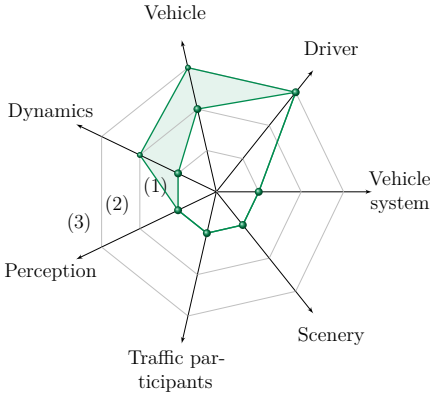


Figure 2.11.: Kiviat digram of DIL simulations [Sch17].

- The remainder of the vehicle is either emulated by, e.g., a dashboard with steering wheels or a real vehicle is incorporated in these simulations.

Motion-based simulators exhibit different levels of fidelity (cf. [Slo08]). Immovable, fixed-based simulators are categorized as low-level simulators. Mid-level simulators are simulators which accelerated just in one degree of freedom (DOF), which is often y-sled, x-sled or a yaw-table. Any force feedback is commonly applied to the steering wheel. High-level driving simulators actuate the payload, e.g., a real vehicle, in at least six DOFs. In many high-level simulators, the vehicle is surrounded by a dome for the projection of the virtual vehicle environment.

2.2.2.4. Hardware-in-the-Loop Simulations

Hardware-in-the-loop (HIL) simulations verify and validate the functionality of the SUT for hardware and software aspects [GRS14]. The real ECU with deployed software implementation of the SUT are verified and validated (cf. model $\mathcal{M}$ in Definition 2.10). The focus of hardware-in-the-loop (HIL) simulations is on the interaction between the SUT with the remaining vehicle components under real-time conditions.
For the interaction with the remainder of the vehicle, the ECU is verified in combination with all connected ECUs or the connected ECUs are simulated by a *rest-bus simulation* (cf. [Sch17]). The *rest-bus simulation* simulates any signals from connect ECUs and the environment.
As system development and vehicle development progresses, HIL simulation setups can be gradually extended to include further vehicle components (cf. [Gie+06]). Hardware components of the vehicle, e.g., sensors and actuators, can also be integrated into HIL simulations or emulated by similar hardware. As shown in Fig. 2.12, the real SUT is verified as ECU in a simulated, emulated, or real vehicle environment. All other dimension of the simulations are still simulated as in MIL and SIL simulations (cf. Sections 2.2.2.1 and 2.2.2.2). HIL simulations offer the flexibility of a MIL and SIL simulations with a higher level of reliability by using the real target hardware (cf. [Gie+06]).

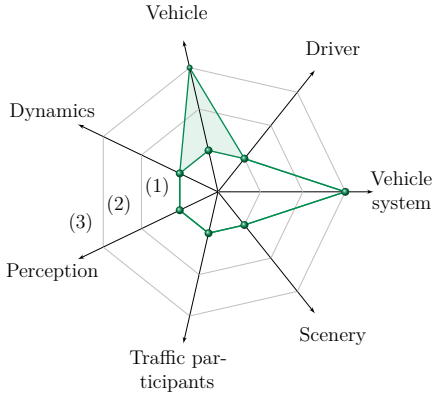


Figure 2.12.: Kiviat digram of HIL simulations [Sch17].

SIL simulations leave the real-time capabilities of the SUT unaddressed because the software implementation is not verified on its target hardware. HIL simulations allow evaluating the real-time behavior of SUT in interaction with the remainder of the vehicles. While SIL simulations can be executed faster or slower than real time, HIL simulations have to be performed in real time because of the real ECUs — including their inputs/outputs — solely operate in real-time. This real-time requirement may require trade-offs for the accuracy and complexity of simulation models as well as specialized real-time platforms for the simulation of the remaining vehicle and environment (cf. [Ise06a]). For example, complex tire and hydraulic models may not be sufficiently calculated in real-time (cf. [Ise06a]).

A variation of HIL simulations are processor-in-the-Loop (PIL) simulations [Neu14]. In PIL simulations, the software implementation of the SUT is executed on a real-time platform (cf. model $\mathcal{M}$ in Definition 2.10). The real-time platform is connected to a host computer which executes the remain models of the control loop in the simulations (cf. Fig. 2.8). The SUT is not executed in real-time but is synchronized with the simulation on the host computer. While HIL simulations use electronic signals for the communication between the ECUs, PIL simulations use direct communication, e.g., Ethernet or Controller Area Network (CAN), for this communication of ECUs.[Kap+16]

Extensions of HIL simulations are vehicle-hardware-in-the-loop (VEHIL) simulations which incorporate the complete vehicle into the simulation loop. VEHIL simulations are described in the following section.

2.2.2.5. Vehicle-Hardware-in-the-Loop Simulations.

A solution for testing full-scale vehicles with all advantages of the previous approaches but without most of their drawbacks is the vehicle-hardware-in-the-loop (VEHIL) simulation (cf. [Gie+06; GRS14]). The complete vehicle with its actuators and sensors is placed on a chassis-dynamometer and is integrated into the simulation (cf. model $\mathcal{M}$ in Definition 2.10). The SUT is either simulated in tools, e.g., automotive data and time-

triggered framework (ADTF) (cf. [Sch07]), emulated on real-time hardware, or execute on the final ECUs within the vehicle. [Sch17; Ste+15]

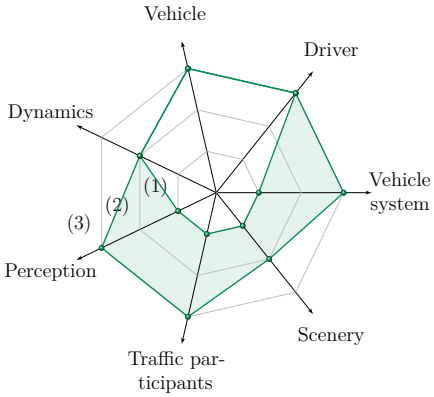


Figure 2.13.: Kiviat digram of VEHIL simulations (cf. [Sch17]).

The vehicle on the chassis-dynamometer is integrated with a simulation of its environment. The dynamometer emulates the vehicle dynamics by emulating the road specific parameters, e.g., slope and friction (cf. Fig. 2.13). Traffic participants are simulated, and their behavior emulated in the vicinity of the real vehicle by movable mobile platforms (cf. [Gie+06; GRS14]). The mobile platform mimics the movement of traffic participants relative to the vehicle on the dynamometer. In rare cases, real traffic participants are integrated in VEHIL simulations (cf. [Sch17]).
The mobile platforms allow for realistic input to the real vehicle sensors which matches the relative position and relative behavior of the corresponding virtual object in the simulation of the environment. The scenery is also simulated and can be emulated. VEHIL are performed in specialized indoor test facilities which allow the control of all environmental parameters, e.g., humidity, ambient light, or temperature [GRS14]. This practice ensures the repeatability of VEHIL simulations.

2.2.2.6. Vehicle-in-the-Loop Simulations

Vehicle-in-the-loop (VIL) simulations are performed with real vehicles on large open test grounds. The vehicle-in-the-loop (VIL) simulation by [BMF07; Boc09] injects simulated environment objects into the sensor perception of the system and project the same objects into the view of the driver using augmented reality. The virtual objects can be supplemented by real objects, e.g., trees or vehicles, as well as emulated objects, e.g., crash targets. As shown in Fig. 2.14, the vehicle dynamics, the vehicle and the driver are real elements. All other dimensions can either be simulated, emulated, or real components.[Sch17]
XIL simulations allow safe, repeatable, and reliable tests of vehicle system under a variety of operation conditions. However, these simulations are inefficient for the verification and validation of interactions between multiple autonomous vehicle systems and the

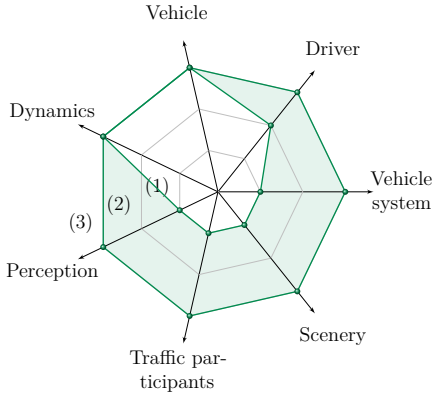


Figure 2.14.: Kiviat digram of VIL simulations [Sch17].

integration of these interactions into the *vehicle systems*. Therefore, in-vehicle tests in the real world are inevitable. [GRS14]

2.2.2.7. Field Operational Tests

Field operational tests with prototype vehicles in real conditions of the real world are the final activity in the validation chain for autonomous vehicle systems. Real world tests can be distinguished into tests with *rapid prototyping* systems and *on-board* tests(cf. [SZ13]). While *rapid prototyping* addresses the verification of the software implementation of the SUT, *on-board tests* are concerned with the complete ECU (cf. [Sch17]). The SUT is either simulation as software in tools, e.g., ADTF (cf. [Sch07]), is emulated on rapid prototyping hardware, or the real ECU is tested in the *field operation tests* (cf. Fig. 2.15).

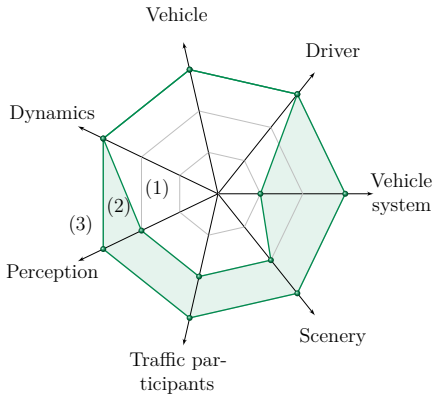


Figure 2.15.: Kiviat digram for real world testing.

For real-world tests, we can distinguish between tests on closed test tracks and tests on public roads. Test tracks allow the adjustment of most environmental conditions but require large test infrastructures (cf. [GRS14]). Besides real objects, e.g., roads and signs, scenery objects, as well as weather conditions, can be emulated. For example, real roads can be rebuild at test grounds in order to safely verify the system behavior on

these roads (cf. Fig. 2.15) [GRS14]. Traffic participants can be other real vehicles and pedestrians or can be emulated by corresponding mock objects, e.g., crash targets. The driver behavior of real vehicle can be emulated using driving robots (cf. [Sch+11]),
Field operational tests on public roads require no dedicated test infrastructure. All encountered objects in the environment are real objects, vehicles, and drivers. However, the reproducibility of field operation tests in public traffic is limited due to the uncontrollability of traffic and the lack of "ground truth" knowledge about the exact states of vehicles in these tests (cf. [Gie+06; GRS14]).
Field operational tests on test tracks and public roads are costly and time-consuming. Complete prototype vehicles with full sensor systems have to be constructed and configured. This heavy engineering of prototype vehicles prohibits the use of field operational tests early in the development cycle.[Gie+06; GRS14]

2.3. Runtime Verification

According to the IEEE [CS217], *verification* encompasses all techniques suitable to ensure that a system satisfies its specification. Traditional verification techniques comprise theorem proving [BC04], model checking [CGP99], and testing [MSB11]. Theorem proving is mostly manually applied and enables the check of the program's correctness as a proof similar to proving the correctness of a theorem in mathematics. Model Checking is an automatic verification technique which is primarily applied to finite-state systems.

Definition 2.30 (Verification)**.** *Verification is formally defined in terms of system behavior as* $\Phi(\mathcal{M}, P, U) \models \psi$ *for given system* $\mathcal{M}$*, given set* P *of parameters, given set* U *of inputs and given property* ψ *which shall hold for the system.[Kap+16]*

Runtime verification represents lightweight verification techniques which complement the three traditional verification techniques [LS09].

Definition 2.31 (Runtime Verification)**.** *Runtime verification encompass verification techniques that allow to check whether a* run *of a system satisfies or violates a given correctness property (cf. [LS09]).*

The main distinction between runtime verification and traditional verification techniques is that runtime verification is predominantly performed at runtime. Executions are the primary objects analyzed in the setting of runtime verification. While traditional verification techniques investigate whether all runs of a system adhere to given correctness properties, runtime verification works on finite (terminated) traces, finite but continuously expanding traces, or on prefixes of infinite traces.[Leu11; LS09]
A *trace* describes a possibly infinite sequence of system's states or system actions. *System states* are formed by the current assignments to system variables or as a sequence of system actions. A trace corresponds to a possibly infinite run of the system. Any

execution of the system is a finite trace and, therefore, is a finite prefix of a run.[Leu11; LS09]

Definition 2.32 (Trace)**.** *A* trace *is a possibly infinite sequence of system's states, which are formed by the current assignments to system variables, or a possibly infinite sequence of system actions [Leu11; LS09].*

Runtime verification approaches vary in the part of a system run which is considered for the verification of the system. Approaches analyze system properties based on the input/output behavior of the system, its state sequences in executions, or based on a sequence of generated events related to the system's execution.[Leu11]
Diagnosis and reconfiguration capabilities can extend runtime verification systems in order to react to property violations and mitigate corresponding system failures (cf. [BLS06; LS09]).
Therefore, runtime verification approaches are distinguished whether they *passively* observe the system's execution and report failures, or whether their monitor verdicts are used to *actively* modify the execution of the target system (cf. [Leu11]).
Runtime verification is closely related to the field of *runtime monitoring*. Monitors evaluate at runtime if the execution of the target system meets a given property. The foundations of runtime monitors are described in the following section.

2.3.1. Runtime Monitor

In runtime verification, *runtime monitors* are used to check whether an execution of a system meets a given property ϕ [BLS11; Leu11; LS09]. In general, *runtime monitors* are automatically generated from given property ϕ.
In mathematical sense, runtime verification checks if a execution w is an element of $[\![\phi]\!]$, where $[\![\phi]\!]$ denotes the set of valid execution for the property ϕ. Thus, runtime verification answers the word problem, i.e., whether a given word is included in some language.[Leu11]

Definition 2.33 (Runtime Monitor)**.** *A Runtime monitor observes an execution of a program or system and evaluates predefined properties, conditions, or invariants about the system behavior based on its observed runtime data (cf. [WH07]).*

The execution trace of a system can be verified *online* or *offline*. The usage of monitors to verify the current execution of a system is denoted as *online monitoring* while the evaluation of recorded traces is defined as *offline monitoring*. *Online monitoring* has the advantage that the critical system behavior can be corrected based on the verdict of the runtime monitor.[LS09]
The complexity for generating the runtime monitors is often negligible because monitors are typically only generated once. However, the complexity of runtime monitors regarding memory and computation requirements are of vital interest because the monitor

should not interfere with the functional and non-functional behavior of the monitored systems.[BLS11]
As part of the trusted computing base, *soundness*, *completeness*, *determinism*, and *passivity* of runtime monitors are of essential importance but are seldom defined and documented in precise terms in the development of runtime monitors (cf. [Fra+17; WH08]). Each characteristic is described in the following:

Soundness : Runtime monitors have to correctly monitor the target systems. Any rejections raised by runtime monitors have to indeed be violations of monitored properties for the target systems.

Completeness: Runtime monitors will sufficiently observe the execution behavior of targets systems for completeness if they detect *all* system behavior violating supervised properties for the target systems.

Determinism: Runtime monitors should yield to the same verification results for identical observations from the execution of target systems.

Passsivity: Runtime monitors should not interfere with the execution of target systems. Passivity is especially critical within embedded systems, where hardware resources are limited.

For observations about the execution of the target systems, parts of the monitoring systems have to be attached to the target systems in order to extract information about the internal operation of these systems. These parts of the monitoring system are termed *probes* or *sensors* (cf. [Sch95; WH07]). Probes (sensors) are further distinguished into *hardware probes* and *software probes*. *Hardware probes* monitor internal physical system signals. *Software probes* are added as additional monitoring code to the code base of the target system in order to logical system signal in the code. The addition of monitoring code to the code base of the target system is termed *code instrumentation* (cf. [Rd04; WH07]).

Definition 2.34 (Code Instrumentation)**.** *Code Instrumentation is the modification of the monitored system by additional code that informs the runtime monitor about events and data relevant for the monitored properties.*

As code instrumentation may interfere with the system execution, runtime monitoring approaches are distinguished in *invasive* and *non-invasive* monitoring [Leu11]:

Invasive monitoring: The runtime monitors of invasive monitoring approaches have an impact on the execution of the systems. These are primarily approaches which use code instrumentation or share the hardware resources with the target system.

Non-invasive monitoring: Non-invasive monitoring approaches segregate the target system from the runtime monitors by using additional computation resources in order to exclude any side effects by the runtime monitors on the execution of the target system.

Other terms for *invasive* and *non-invasive* monitoring are *intrusive* and *non-intrusive* monitoring (cf. [Sch95]).
Three types of runtime monitoring are distinguished; software, hardware, and hybrid monitoring (cf. [WH07]). Each type is discussed in the following sections.

2.3.1.1. Software Monitors

Software monitoring requires the modification of the targeted systems. The primary approach is the instrumentation of the system's application code by additional code that informs the runtime monitor about relevant events and data values. Further approaches are the instrumentation of the operating systems or the usage of a dedicated monitor process (cf.[HG08; WH07]).
Software monitoring can be used flexibly and does not require any additional hardware (cf. [Rd04; Sch95; Tha+03]). However, the integration of runtime monitoring code into the target system may introduce side effects for the target system. The software monitoring may introduce overhead on the system's processing time and memory space consumption. Such side effects can be unacceptable for many real-time systems. An alternative approach is to introduce the instrumentation code permanently into the system, but this may be difficult for an embedded system with strict costs restrictions. [WH07]
Instead of introducing instrumentation code to the target system, additional hardware components can be introduced for the runtime monitors. Hardware monitoring is described in the following section.

2.3.1.2. Hardware Monitors

Hardware monitoring is machine dependent resp. processor architecture dependent. It comprises the modification of the hardware platform on which the target program is executed. Either internal signals of the target system are probed, or additional monitoring hardware is introduced to the hardware platform. The monitoring hardware observes the execution of the target program by, e.g., listening on the system bus. Instrumentation of system code is not required for hardware monitoring. [Sch95; WH07]
The benefit of hardware monitoring is the non-intrusive access to information about the target system. Therefore, hardware monitoring is most suitable for a system with strict real-time constraints. [WH07]
The disadvantage of hardware monitoring is the increasing costs and inherent limitations of monitoring hardware (cf. [Sch95; WH07]). Newer hardware platforms are less likely to offer possibilities for physical probe points. System-on-chip (SoC) systems with on-chip caching and complex processing and memory architectures have significantly reduced the visibility of program execution for external hardware, e.g., runtime monitoring systems. A solution to the reduced external observability of SoC systems are on-chip monitors (cf. [El 02; WH07]).
Software and hardware monitoring can be combined with hybrid monitoring in order to mitigate their disadvantages. Hybrid monitoring is described in the next section.

2.3.1.3. Hybrid Monitors

Hybrid monitoring encapsulates monitoring approaches which combine software and hardware monitoring. Hybrid monitoring tries to combine the advantages of each approach by simultaneously mitigating their disadvantages. Code instrumentation of the target system allows access to internal information about the system execution by defining relevant events for the runtime monitoring within the code of the target system. These events would not be accessible by hardware monitors. The instrumentation code is executed as a part of the target system, and relevant events are transferred to separate monitoring hardware for event detection and event processing The additional instrumentation code may potentially affect the behavior and performance of the target systems, but the usage of hardware probes reduces the amount of instrumentation code which is required to monitor these systems sufficiently. As results, hybrid monitors are less likely to interfere with the execution of the target system. [Rd04; Sch95; Tha+03; WH07]
Besides runtime verification, *runtime monitoring* is used in other contexts for various system types, e.g., safety-critical and mission-critical systems, enterprise systems software, autonomous systems, reactive control systems, health management systems, diagnosis systems, and system security and privacy with various purposes, e.g., debugging, testing, security, safety monitoring, verification, validation, faults protections, profiling, or behavior modification.[Rd04; Sch95]
For the monitoring and verification of systems at runtime, relevant system properties have to be formally defined in the system development. The following section gives an overview of the formal definition of such system properties.

2.3.2. Property Specification

Runtime Verification generally assumes a (formal) logic for the description of properties which are expected to be satisfied by the monitored system [Fra+17]. Havelund and Goldberg define in [HG08] four dimensions for the specification of the monitored system behavior:

Location quantification: Location quantification addresses if a logic allows quantifying over locations in the monitored systems. Runtime monitors evaluate when the system reaches a specific system location during its execution. For online verification, these locations contain the monitoring code itself. For offline verification, these locations contain code which will generate events and forward them to the external runtime monitor. An example of location quantification are system invariants, state machine notations, and process algebras.[HG08]

Temporal quantification: Temporal quantification addresses if a logic allows for quantification over time points. For example, whether *open* and *close* methods for access to files are called in their respective order. Such logic can either state ordering relationships over events and actions or reason about relative or absolutes time values and duration for over events and system actions. Temporal logics, e.g.,

linear temporal logic (LTL) [Pnu77], and temporal automata, like used in UPPAAL [LPY97], exist specifically for the temporal quantification over system states.

Data quantification: Data quantification addresses whether a logic allows the binding and referral to values across states in a forward or backward direction in time. For example, the calling of *open* and *close* methods for the same file includes the data binding over the file. Data quantification presumes temporal quantification. Specific data logics are defined for data quantification. [HG08]

Abstract data specification: Abstract data specification consider logics which are able to abstract concrete program states to abstract specification states. General purpose specification languages, e.g., abstract state machine language (ASML) [Gri+01] or Maude [Cla+99], exist which inherently support abstract data specification. These languages commonly have executable subsets which resemble a functional or state-based programming language.[HG08]

The expressiveness of logic is essential for runtime verification. Specification logics must only define system properties which can be verified for the (finite) trace exhibit for the execution of the system. Properties about multiple or infinite system executions are not suitable for runtime verification.[Fra+17]
The following section introduces the formal logic which is used in this thesis for the definition of monitored system properties.

2.4. Typed First-Order Logic

The definition of system properties for the runtime monitoring in this thesis uses a *typed first-order logic*. The type restrictions of variables, functions, and predicates reduce the processing complexity of the runtime monitoring. The evaluation of variable, functions, and predictions by the runtime monitoring has exclusively considered objects with matching types. All other objects can be neglected. The following sections introduce the typing, syntax, and semantics of this *typed first-order logic*. The typed first-order logic has been first introduced in [BHS07].

2.4.1. Types

The major difference between the *typed first-order logic* and traditional first-order logic (cf. [Smu12]) is the explicit *typing* of objects, variables, functions, predicates. Typing of first-order logic has been defined by Beckert et al. in [BHS07]. The typing allows to reason about the domain of the runtime monitoring on the abstract level of *types* instead of individual real world objects. Every object of the domain is assign to one *type*. *Type* is defined as "a category of [...] things having common characteristics" [Ste10]. All *types* for a particular problem domain are organized in a *type hierarchy*:

Definition 2.35. *A type hierarchy is a quadruple* $\mathbb{T} = \langle \mathcal{T}, \mathcal{T}_D, \mathcal{T}_A, \sqsubseteq \rangle$ *consisting of*

- *a set of* types $\mathcal{T}$*,*
- *a subset of* abstract types $\mathcal{T}_A$*,*
- *a subset of* dynamic types $\mathcal{T}_D$*, and*
- *a relation* $\sqsubseteq$ *denoting the sub-type relationship between types.*

It holds that $\mathcal{T}_A \cup \mathcal{T}_D = \mathcal{T}$ *and* $\mathcal{T}_A \cap \mathcal{T}_D = \emptyset$*. The sub-type relation* $\sqsubseteq$ *is a reflexive partial order on* $\mathcal{T}$ *(cf. [BHS07]). We say that type* A *is a subtype of* B *if* $A \sqsubseteq B$*. Every abstract type* $B \in \mathcal{T}_A \setminus \{\bot\}$ *has a non-abstract subtype:* $A \in \mathcal{T}_D$ *with* $A \sqsubseteq B$*. There is an empty type* $\bot \in \mathcal{T}_A$ *and a universal type* $\top \in \mathcal{T}_D$ *with* $\bot \sqsubseteq z \sqsubseteq \top$ *for all* $z \in \mathcal{T}$ *(cf. [BHS07]). The set without the empty type* $\bot$ *is denoted by* $\mathcal{T}_Q := \mathcal{T} \setminus \{\bot\}$*.*

Typing also applies to variables, functions, and predicates of the *typed first-order logic*. Variables and parameters of function and predicates can be restricted to specific types. Objects will only be considered for a typed variable or parameters of typed functions and typed predicates if the objects match the specific types or their sub-types.
The typing of the *typed first-order logic* is aligned with the implicit typing of requirements and explicit typing of runtime data in autonomous vehicle systems. For the runtime monitoring of autonomous vehicles, requirements are transformed into *typed first-order logic*, and the types in these requirements build the *type hierarchy* $\mathbb{T}$ of the *typed first-order logic*. The *typed first-order logic* is then evaluated at runtime on the provided typed data from the autonomous vehicle systems.
The definition of variables, functions, and predicates by the signature of the *typed first-order logic* is described in the following section.

2.4.2. Signature

The *typed first-order logic* has the signature $\Sigma = \langle V, F, P, \alpha \rangle$ with a set of variable symbols V, a set of function symbols F, a set of predicate symbols P with an arity of $n > 0$, and a typing function α (cf. [BHS07]). Variables enable to consider different objects and entities of the autonomous vehicle systems and their environments while functions and predicates reason about particular properties of these objects and parameters and their relations.

Definition 2.36. *The signature* Σ *for a given type hierarchy* $\mathbb{T} = \langle \mathcal{T}, \mathcal{T}_D, \mathcal{T}_A, \sqsubseteq \rangle$ *is quadruple* $\Sigma = \langle V, F, P, \alpha \rangle$ *of*

- *a set of variable symbols* V*,*
- *a set of function symbols* F *with an arity of* $n \geq 0$*,*
- *a set of predicate symbols* P *with an arity of* $n > 0$*, and*
- *a typing function* α*.*

We denote the arity k of a function with f^k and for predicate with p^k. A function with an arity of 0 *is also called a* constant. *Logical symbols, like $\wedge, \vee, =, \neq, \forall, \exists$ are defined as usual*

Definition 2.37. *The symbol α denotes the typing function that assigns every symbol of V, F, and P a corresponding type. Therefore, variables, functions, and predicates are only applicable for this particular type (cf. [BHS07]):*

$$\begin{aligned}
&\alpha(x) \in \mathcal{T}_Q \text{ for all } x \in V \\
&\alpha(f) \in \mathcal{T}_Q^* \times T_Q \text{ for all } f \in F \\
&\alpha(p) \in \mathcal{T}_Q^* \text{ for all } p \in P \\
&\text{with } \mathcal{T}_Q := \mathcal{T} \setminus \{\bot\}.
\end{aligned}$$

Consider: We require all parameters of functions and predicate to be from $\mathcal{T}_Q$ and therefore exclude the empty type $\bot$.

There exist predicate symbol $\alpha(\doteq) = (\top, \top)$ for the type equality and predicate symbol $\mathbb{E}z \in P$, which is called type predicate for type z, with typing $\alpha(\mathbb{E}z) = (\top)$ for any $z \in \mathcal{T}$. A function symbol (z) represents the type cast of the static type of an expression $\alpha((z)) = ((\top), z)$ for each $z \in \mathcal{T}_q$. The dynamic type of the expression remains unchanged. The type cast enables the usage of sub-types as arguments for functions and predicates. The following notions are defined (cf. [BHS07]):

$$\begin{aligned}
&v\!:\!z \text{ for } \alpha(v) = z \text{ with } v \in V, z \in \mathcal{T}_Q \\
&f\!:\!z_1, \ldots, z_n \to z \text{ for } \alpha(f) = ((z_1, \ldots, z_n), z) \; for \; z_i, z \in \mathcal{T} \\
&p\!:\!z_1, \ldots, z_n \text{ for } \alpha(p) = (z_1, \ldots, z_n) \; for \; z_i \in \mathcal{T}
\end{aligned}$$

2.4.3. Terms and Formulas

The expression of requirement conditions in the *typed first-order logic* requires the definition of *terms* and *formulae* over the variables, function, and predicate symbols.

Definition 2.38. *Based on the signature $\Sigma = \langle V, F, P, \alpha \rangle$ we can inductively define the set of terms $Term_z$ for each type $z \in \mathcal{T}$ (cf. [BHS07]):*

- *$x \in Term_z$ for any variable $x\!:\!z \in V$,*
- *$f(t_1, \ldots, t_n) \in Term_z$ for any function symbol $f(z_1, \ldots, z_n) \to z \in F$ and terms $t_i \in Term_{z_i'}$ with $z_i' \sqsubseteq z_i$ for $1 \leq i \leq n$*

All entities built in the described way are terms and no others. The static type for any term $t \in Term_z$ is written as $\sigma(t) := z$. The set of terms $Term_{\mathcal{T}}$ over the signature $\Sigma = \langle V, F, P, \alpha \rangle$ and a set of Types $\mathcal{T}$ is then denoted by $Term_{\mathcal{T}} := \bigcup_{z \in \mathcal{T}} Term_z$. A term without any variables is called a ground term.

Conditions of requirements which are considered for the runtime monitoring can be formally expressed as *first-order formulae*.

Definition 2.39. *The set of first-order formulae Fml over the signature $\Sigma = \langle V, F, P, \alpha \rangle$ are defined as followed:*

- $True, False \in Fml$ *are formulae.*
- $p(t_1, \ldots, t_n) \in Fml$ *for a predicate symbol* $p(z_1, \ldots, z_n)$ *and terms* $t_i \in Term_{z_i'}$ *with* $z_i' \sqsubseteq z_i$ *for all* $1 \leq i \leq n$ *is a formula.*
- *Let* t_1 *and* t_2 *are terms, then* $(t_1 \doteq t_2)$ *is an (atomic) formula.*
- *Let* ϕ *and* ψ *be formulae, then* $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \phi \rightarrow \psi$ *are formulae.*
- *Let* ϕ *be a formula and* $v \in V$ *a variable, then* $\exists v.\phi, \forall v.\phi$ *are formulae.*

We call ϕ the scope of the variable $v \in V$ and say that v is bound by the quantifiers $\forall, \exists$ in the formulae $\forall v.\alpha$ resp. $\exists v.\alpha$. For syntax sugaring we write $t \sqsubseteq z$ instead of $\sqsubseteq z(t)$ with $z \in \mathcal{T}$ $t \in Term_{\mathcal{T}}$ as well as $t_1 \doteq t_2$ instead of $\doteq (t_1, t_2)$ for $t_i \in Term_{\mathcal{T}}$. The usual parentheses and precedence rules apply.

The *quantifiers* $\forall$ (universal) and $\exists$ (existential) bind all occurrences of variable x in the sub-formula ϕ. The formula ϕ is also called the *scope* of the quantified variable x. Occurrences of variables which are not bound by any quantifier are called *free*. A formula ψ without any free variables is called a *closed formula*

Definition 2.40. *The set of free variable $FV(t)$ for a term t is defined by*

- $FV(t) = \{v\}$ *for* $v \in V$, *and*
- $FV(f(t_1, \ldots, t_n)) = \bigcup_{i=1,\ldots,n} FV(t_i)$.

The set of free variables for a formula is then defined by

- $FV(p(t_1, \ldots, t_n)) = \bigcup_{i=1,\ldots,n} FV(t_i)$,
- $FV(t_1 \doteq t_2) = FV(t_1) \cup FV(t_2)$,
- $FV(true) = FV(false) = \emptyset$,
- $FV(\neg\phi) = FV(\phi)$,
- $FV(\phi \wedge \psi) = FV(\phi \vee \psi) = FV(\phi \rightarrow \psi) = FV(\phi) \cup FV(\psi)$, *and*
- $FV(\forall x.\phi) = FV(\exists x.\phi) = FV(\phi) \setminus \{x\}$.

A formula ϕ is then called closed iff $FV(\phi) = \emptyset$.

The definition of *terms* and *formulae* solely address the *syntax* of the *typed first-order logic* but not their interpretation. The interpretation of *terms* and *formulae* by the *semantics* of the *typed first-order logic* is described in the following section.

2.4.4. Semantics

First-order logics have to be evaluated based on models $\mathcal{M}$ (also called *structure*) which encompass the concrete objects, functions, and predicates over which the terms and formulae are evaluated (cf. [BHS07]).

Definition 2.41. *The model for a typed first-order logic with the types system $\mathcal{T}$ and signature $\Sigma = \langle V, F, P, \alpha \rangle$ is defined as $\mathcal{M} = \langle \mathbb{D}, \delta, \mathcal{I} \rangle$ with*

- *the domain $\mathbb{D}$,*
- *the dynamic typing function δ, and*
- *the interpretation of functions and predicates $\mathcal{I}$.*

We further define $\mathbb{D}_z := \{d \in \mathbb{D} \mid \delta(d) \sqsubseteq z, z \in \mathcal{T}\}$ as the set of parameters and objects of type $z \in \mathcal{T}$. Opposed to [BHS07], this thesis does not require the set $\mathbb{D}_z$ to be a non-empty set for all $z \in \mathcal{T}_D$ because there might exits no objects for some types in the runtime monitoring. For example, there might be no other vehicles in the environment of autonomous vehicle systems. Therefore, the objects set for the type *vehicle* would be empty — considering that the *automated* ego *vehicle* is a distinctive type.
The typing function $\delta : \mathbb{D} -> \mathcal{T}_D$ assigns every object of domain $\mathbb{D}$ a dynamic type $\mathcal{T}_D$. In case no particular type has yet been defined for a domain object because it has not yet been known, the typing function $\delta : \mathbb{D} -> \mathcal{T}_D$ assigns the universal type $\top$ to this object.

Definition 2.42. *The interpretation $\mathcal{I}$ assigns every function symbol $f \in F$ and predicate symbol $p \in P$ a concrete function resp. predicate over the parameters and objects with the correct typing $z \in \mathcal{T}$. It holds that;*

- *for any $f : z_1, \ldots, z_n \to z \in F$, $\mathcal{I}$ yields a function $\mathcal{I}(f) : \mathbb{D}_{z_1} \times \cdots \times \mathbb{D}_{z_n} \to \mathbb{D}_z$ with $z, z_i \in \mathcal{T}, i \in \mathbb{N}$*
- *for any $p : z_1, \ldots, z_n \in P$, $\mathcal{I}$ yields a subset $\mathcal{I}(p) \subseteq \mathbb{D}_{z_1} \times \cdots \times \mathbb{D}_{z_n}$ with $z_i \in \mathcal{T}, i \in \mathbb{N}$*
- *for type casts, $\mathcal{I}((z))(x) = x$ if $\delta(x) \sqsubseteq z$, otherwise $\mathcal{I}((z))(x)$ is an arbitrary but fixed element of $\mathbb{D}_z$.*
- *for equality $\mathcal{I}(\doteq) = \{(d, d) \mid d \in \mathbb{D}\}$,*
- *for type predicates, $\mathcal{I}(\sqsubseteq z) = \mathbb{D}_z$ with $z \in \mathcal{T}$.*

A model $\mathcal{M}$ is not sufficient to completely reason about arbitrary terms and formulae because the model does not address the variables of the *typed first-order logic*. Therefore, the notion of *variable assignments* are introduced:

Definition 2.43. *For a model $\mathcal{M} = \langle \mathbb{D}, \delta, \mathcal{I} \rangle$, a variable assignment is a function $\beta : V \to \mathbb{D}$, such that*

$$\beta(x) \in \mathbb{D}_z \text{ for all } x : z \in V.$$

The modification β_x^d *of a variable assignment* β *for any variable* $x:z$ *and any element* $d \in \mathbb{D}_z$ *of the domain with type* $z \in \mathcal{T}$ *is defined as*

$$\beta_x^d(y) := \begin{cases} d, & \text{if } y = x \\ \beta(y), & \text{otherwise} \end{cases}$$

The definitions of model $\mathcal{M}$ and variable assignment β enable to formally define the *semantics of terms*. A *evaluation function* $\text{val}_\mathcal{M}$ is introduced for the interpretation of terms.

Definition 2.44. *For a model* $\mathcal{M} = \langle \mathbb{D}, \delta, \mathcal{I} \rangle$*, and a variable assignment* β*, the valuation function* $val_\mathcal{M}$ *can be inductively defined by*

- $val_{\mathcal{M},\beta}(x) = \beta(x)$ *for any variable* $x \in V$*.*
- $val_{\mathcal{M},\beta}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(val_{\mathcal{M},\beta}(t_1), \ldots, val_{\mathcal{M},\beta}(t_n))$ *with terms* t_1, t_n*.*

For a ground term t*, one can simply write* $val_\mathcal{M}(t)$*, because* $val_{\mathcal{M},\beta}(t)$ *is independent of* β

The *semantics of formulae* is defined by their validity. The validity of a formula for a given model $\mathcal{M}$ and a given variable assignment β is determined by the *validity relation* $\models$.

Definition 2.45. *The validity relation* $\models$ *is inductively defined for a given model* $\mathcal{M} = \langle \mathbb{D}, \delta, \mathcal{I} \rangle$*, and a variable assignment* β *by*

- $\mathcal{M}, \beta \models p(t_1; \ldots, t_n)$ *iff* $(val_{\mathcal{M},\beta}(t_1), \ldots, val_{\mathcal{M},\beta}(t_1)) \in \mathcal{I}(p)$*.*
- $\mathcal{M}, \beta \models true$
- $\mathcal{M}, \beta \not\models false$
- $\mathcal{M}, \beta \models \neg\phi$ *iff* $\mathcal{M}, \beta \not\models \phi$
- $\mathcal{M}, \beta \models \phi \wedge \psi$ *iff* $\mathcal{M}, \beta \models \phi$ *and* $\mathcal{M}, \beta \models \psi$
- $\mathcal{M}, \beta \models \phi \vee \psi$ *iff* $\mathcal{M}, \beta \models \phi$ *or* $\mathcal{M}, \beta \models \psi$*, or both*
- $\mathcal{M}, \beta \models \phi \rightarrow \psi$ *iff*$\mathcal{M}, \beta \models \phi$ *then also* $\mathcal{M}, \beta \models \psi$
- $\mathcal{M}, \beta \models \forall x.\phi$ *(for a variable* $x:z$*) iff* $\mathcal{M}, \beta_x^d \models \phi$ *for every* $d \in \mathbb{D}_z$
- $\mathcal{M}, \beta \models \exists x.\phi$ *(for a variable* $x:z$*) iff there exists some* $d \in \mathbb{D}_z$ *such that* $\mathcal{M}, \beta_x^d \models \phi$

If $\mathcal{M}, \beta \models \phi$*, we say that* ϕ *is valid in the model* $\mathcal{M}$ *under the variable assignment* β*. For a closed formula* ϕ*, we write* $\mathcal{M} \models \phi$*, since* β *is then irrelevant.*

The validity of formulae can be generalized as a general property of formulae concerning arbitrary models and arbitrary variable assignments.

Definition 2.46. *Under a fixed but arbitrary type hierarchy* $\mathbb{T}$*, and signature* Σ*,*

- *A formula* ϕ *is logically valid if* $\mathcal{M}, \beta \models \phi$ *for any model* $\mathcal{T}$ *and any variable assignment* β*.*
- *A formula* ϕ *is satisfiable if* $\mathcal{M}, \beta \models \phi$ *for some model* $\mathcal{T}$ *and some variable assignment* β*.*
- *A formula* ϕ *is unsatisfiable if it is not satisfiable — if* $\mathcal{M}, \beta \not\models \phi$ *for any model* $\mathcal{T}$ *and any variable assignment* β*.*

The following chapter presents and analyzes the current development process for autonomous vehicle systems in the automotive domain. The *typed first-order logic* is considered in the definition and implementation of the runtime monitoring framework in Chapter 6.

3. Problem Outline

This chapter presents the current development practice for autonomous vehicle systems in the automotive domain on the example of an industrial lane change assistant. An overview of the functionality of the lane change assistant and its development process is given in Section 3.1. The individual activities in the development of the lane change assistant are described in more detail Sections 3.2 to 3.7. The chapter ends with an assessment of limitations for the current development practice in Section 3.8

3.1. Running Example: Lane Change Assistant

In this work, we examine a lane change assistant as a running example. The lane change assistant is part of an industrial highway pilot, which is widely considered the next development iteration of autonomous vehicle systems towards full automated vehicles (cf. Fig. 2.4). Vehicles with activated highway pilot will autonomously drive in traffic on highways towards their assigned destination. The vehicle automation includes the autonomous execution of lane changes.

3.1.1. Basic Functionality of Lane Change Assistant

Lane change assistants address the guidance of the vehicle on tactical level (cf. Fig. 2.3). The main functionality of a lane change assistant is to perform lane changes to the left or right adjacent neighbor lane under consideration of the current traffic situation. The lane changing includes changes between lanes on multi-lane highways as well as changes to on- and from off-ramps in order to enter resp. exit highways. Another functionality is the safe merging in weaving areas of interchanges or front of road constrictions. The lane change assistant has to evaluate if lane changes can be safely executed and if they are beneficial for the progress of the trip in all these use cases. There exist many reasons for the execution of lane changes. Some lane changes are beneficial, e.g., overtaking slower vehicles, (cf. Fig. 3.1 [BMW18]), while other lane changes might be inevitable. For example, a lane change will be inevitable if the current route requires the vehicle to leave or change the highway.
In summary, a lane change assistant has to perform safe lane changes autonomously.
The safety of lane changes for the automated ego vehicle, its passengers and other traffic participants is essential. Any risks for any traffic participants have to be excluded. For the execution of lane changes, the lane change assistant has to consider national traffic codes and the traffic situations in its vicinity. The assistant has to perceive the road infrastructure, e.g., road lanes and their markings, as well as other traffic participants, e.g.,

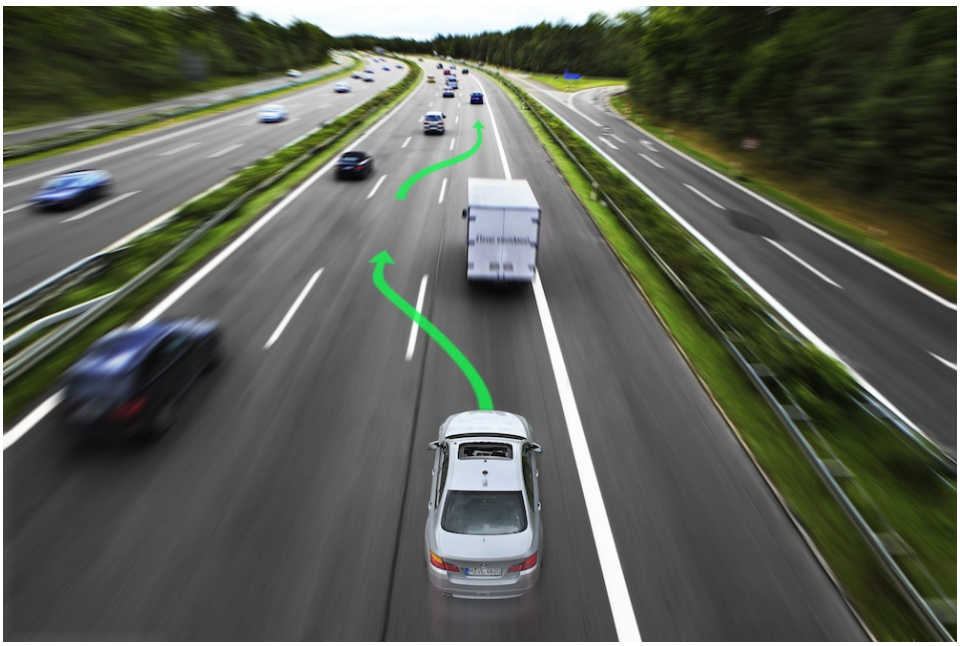


Figure 3.1.: Lane Change Assistant [BMW18].

other vehicles or pedestrians. Based on the relative positions and velocities of other traffic participants, the lane change assistant determines if a lane change to the left resp. right adjacent neighbor lane is possible without endangering other traffic participants. Other traffic participants should not be forced to brake hard or to perform emergency maneuvers. Additionally, the lane change assistant has to obey national traffic codes, e.g., signs or road markings prohibiting lane changes, in order to avoid any maneuvers which would be unexpected or irrational for other traffic participants.
Passengers expect the lane change assistant to inform them about all its decisions and considered information about the vehicle and its vicinity. HMIs have to present this information in a clear and understandable manner without overstraining drivers. Human drivers expect to be able to take over the vehicle control at any time — especially in critical situations which the lane change assistant misinterpreted. Therefore, the driver must be able to overrule the decisions made by the lane change assistant.
As part of the highway pilot, the lane change assistant is only developed for highways. Rural and urban roads are not yet considered for the functionality of the lane change assistant due to their increased situational complexity (cf. [CC04]). Further improvements of the lane change assistant may lead to its application to these domains.
The following section gives an overview of the development activities for the lane change assistant, as they were performed in the case study (cf. Chapter 8).

3.1.2. Development Activities

In the automotive domain, the basic development process follows the V-model consisting of a design and implementation part and a verification and validation part (cf. [RB08]).

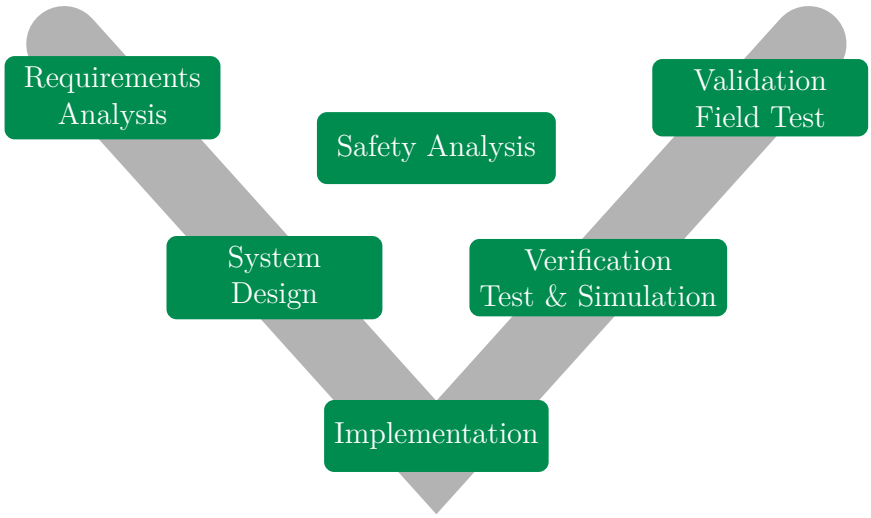


Figure 3.2.: Activities in the development, verification and validation of the lane change assistant.

The activities for the development of the lane change assistant follow this development process (cf. Fig. 3.2). Sections 3.2 to 3.7 describe each development activity in more detail.
As shown in Fig. 3.2, the first activity in the development process of the lane change assistant is the *requirements analysis*. in the requirement analysis, requirements from relevant stakeholders are elected for the *system specification* of the lane change assistant (cf. Definition 3.2) . The specification is the first documented definition of the lane change assistant's functionality and restrictions. The majority of requirements for the lane change assistant emerge from the development of the superior highway pilot.
In the *system design* (cf. Fig. 3.2), the lane change assistant is designed based on the requirements in the *system specification* by defining an architecture of functional components. Each component addresses one or more basic functions required by the system specification of the lane change assistant. In the course of the development, the system specification is further detailed for its subsystems. A technical concept defines each component, the algorithms and communication techniques used for realizing the components' functionality. The lane change assistant is divided into the processing of sensor data for the *scene* representation, the situation assessment of the *scene* representation and the behavior planning for lane changes. Additional to the functional architecture, a hardware architecture is defined incorporation a network of ECUs for the execution of the functional components as well as sensor and actuator hardware components for the perception and interaction with the vehicle's environment.
The system specification and system architecture allow analyzing the proposed system under safety aspects. The *safety analysis* as third activity in the development process (cf. Fig. 3.2) evaluates the concept of the lane change assistant from the *system design* in different operation modes. A operation mode encompasses the states of software functions, hardware components, and the environment of the system — the real world. Critical operational modes are identified in which the behavior of the lane change assistant

impose a potential safety risks for its passengers, other vehicles, and their passengers. For each critical operation mode, safety measures are defined (cf. Definition 3.1). The safety measures are intended to mitigate the safety risks of the lane change assistant in the operation mode. These measures are incorporated into the system development as additional requirements. These safety requirements are added to the system specification resulting in a revision of the system design and its components. The correct and effective implementation of all identified safety requirements, including their measures, have to be addressed in the verification and validation. In the automotive domain, the safety analysis commonly proceeds following the ISO 26262 [Int09d].

Definition 3.1 (Safety Measure)**.** *"The activity or technical solution to avoid or control systematic failures and to detect random hardware failures or control random hardware failures, or mitigate their harmful effects" [Int09a].*

Following the revision of requirements and system design, the lane change assistant is implemented. In the *implementation* activity (cf. Fig. 3.2), the functionality and safety measures of each functional component are primarily realized in software and in minor cases solely as hardware. For the lane change assistant, the atomic functional components — the lowest components in the hierarchy of the functional system architecture — are refined as software-components and implemented as C/C++ code. The code is written manually or generated from models, e.g., signal flow diagrams using modeling tools e.g., Matlab/Simulink (cf. [Bis96]) or Ascet (cf. [Lef+97]). The correct implementation of each atomic functional component is addressed in unit tests. Each implementation artifact of a atomic component — e.g., its classes and functions — is independently stimulated by the test inputs of corresponding test cases in order to reveal bugs and faults (cf. Section 2.2). The response of the implementation artifact — its output data — is compared with the envisaged response of the executed test case. Additional hardware required for the functional components, e.g., sensors or ECUs, are realized and tested simultaneously.
Following the system architecture, atomic implementation artifacts are integrated into compositional components. Compositional components might be further integrated with other atomic or compositional components. This way larger parts of the system are realized. The integration of functional components may lead to the emergence of more complex functionality. For example, the combination of sensor preprocessing and modeling enables the system to generate an internal representation of the current vehicle environment.
In the *verification* activity (cf. Fig. 3.2), the functional correctness of composite components is verified in *integration tests* with regard to the requirements of the system specification. Integration tests presume the correct implementation of each subordinate component — as it is verified in unit tests for atomic functional components — and therefore focus on the communication and interaction between subordinate components and emerging functionalities. For the lane change assistant, simulations are used for integration testing — from MIL, SIL, HIL to VIL (cf. Section 2.2.2). These simulation methods are differentiated based on the SUT — e.g., if it is a model, implementation, or

the real system or functions — and the incorporation of the target hardware. For all simulation methods, test cases resp. test scenarios define a set of stimuli for which the compositions are verified. The responses of compositions to these stimuli are compared to the expected response in the test case. At the end of the hierarchical integration, the complete lane change assistant is assembled and verified. The integration tests of the assembled system are also referred to as *system test*.
Additional to the verification in tests and simulations, formal verification methods, e.g., model checking or static code analysis (cf. [Alu15]), are adopted at times.
Besides the verification of the lane change assistant in system tests, the system has to be validated. The *validation* activity is the last development activity for the lane change assistant before its deployment and commercialization. For validation, stakeholders test the system for the satisfaction of their requirements. The validation also addresses implicit requirements, which have not yet been identified and documented in the requirements analysis. Tests performed by stakeholders are also referred to as *acceptance tests*. These acceptance tests are commonly performed with prototype vehicles in the real world to mimic the experience of later customers.
In the following sections, each development activity is described in the context of the case study — the development of the lane change assistant (cf . Chapter 8).

3.2. Requirements Analysis

The basic functionality of the lane change assistant, which has been described in Section 3.1.1, yields from requirements of involving stakeholders (cf. Definition 3.2). These requirements are gathered and improved in interviews with each stakeholder in the requirements analysis (cf. Section 3.1.2) . The analysis addresses the unambiguousness, quantification, and completeness of these requirements.

Definition 3.2 (Requirement)**.** *"A statement that identifies a necessary attribute, capability, characteristic, or quality of a system in order for it to have value and utility to a user" [You01].*

All requirements gathered in the requirements analysis are documented in the *system specification* (cf. Definition 3.3). The requirements are commonly described in natural language and are hierarchically structured where each subordinate requirement further refines its superordinate requirements. Requirements addressing similar concerns are grouped. For the lane change assistant, the system specification has been provided by a project partner.

Definition 3.3 (System Specification)**.** *A documented set of mandatory requirements for a system [Int17].*

The system specification for the lane change assistant and its requirements are presented in the following sections under consideration of the international standard ISO/IEC

25010:2011 [Int11b]. The standard ISO/IEC 25010:2011 defines a quality model for software products with eight categories: functionality, reliability, usability, efficiency, maintainability, compatibility, security, and portability. We use these categories as guidance for the presentation of the provided requirements in the following sections.

3.2.1. Functionality

As described in Section 3.1.1, the basic functionality of the lane change assistant is to perform lane change on highways. In its current development state, rural and urban roads are not considered for the lane change assistant due to differences in the complexity of environment situations (cf). Lane changes on highways can be further subdivided into two use cases; lane changes between ongoing highway lanes and lane changes from on-ramps resp. to off-ramps in order to enter resp. exit highways. Additional functional requirements exist for the lane change assistant. These requirements are applicable for either use case. The functional requirements for the lane change assistant are presented in more detail in the following sections.

3.2.1.1. Lane Changes between Highway Lanes

The major use case for the lane change assistant is the changing between lanes on multi-lane highways (cf. FR_1_1 in Table 3.1). For this use case, the assistant has to determine the current driving domain (FR_1). The lane change assistant is only developed for highways as part of a highway pilot and is currently unable to safely cope with the complexity of traffic in other domains, e.g., *rural* and *urban* roads.
On highways, the lane change assistant has to independently change from the velocity and available gap sizes between vehicles. The assistant has to perform lane changes in traffic jams with small gaps between vehicles and at slow velocities (FR_1_2) as well as in normal traffic with high differences in their velocities but large gaps between vehicles (FR_1_1). In normal traffic, the automated vehicle must drive faster than 10 m/s (FR_2_1). Otherwise, the overtaking would take too much time and compromise other traffic participants. Additionally, a lane change is not supposed to be executed in curves with a curvature less than 125 m because actuators might not be able to execute lane changes safely (cf. [Vis+08]). The automated vehicle would leave the target lane (FR_2_2). Before every lane change, the automated vehicle has to be properly aligned inside its current lane in order to guarantee the correct and safe execution of the lane changes. A lane change is not supposed to be executed if the position of the automated ego vehicle deviates more than 0.4 m from the center of its current lane (FR_2_3).
A special case for lane changes on multi-lane highways is the merging with other traffic participants in front of constrictions. In general, the traffic in front of constrictions is slow and dense because the traffic of two or more lanes has to be condensed to fewer lanes. A lane change is inevitable for the automated vehicle if it drives on a lane with the constriction. As stated in requirement FR_4 (cf. Table 3.1), the automated vehicle must be able to perform alternate merging starting less than 100 m in front of the constriction. The constrictions limit the time and space available for the lane change.

Table 3.1.: Requirements for the lane changing on highway.

ID	Description
FR_1	The system shall be able to detect if the current domain is a highway.
FR_1_1	The system shall be able to perform lane changes on multi-lane highways.
FR_1_2	The system shall be able to change lanes in a traffic jam with a traffic flow velocity of less than 5 m/s and gaps of less than 10 m between vehicles.
FR_2	The system has to prevent unsafe lane changes.
FR_2_1	The system shall be able to prevent lane changes if the ego velocity is less than 10 m/s.
FR_2_2	The system shall be able to prevent lane changes if the curvature of the ego lane is less than 125 m.
FR_2_3	The system shall be able to prevent lane changes if the lateral offset to the center of the current lane is more than 0.4 m.
FR_3	The system shall be able to adjust the vehicle longitudinally towards a suitable gap in a search range of 100 m to the front and 100 m to the rear of the vehicle in order to prepare a lane change.
FR_4	The system shall be able to perform lane changes with less than 100 m in front of constrictions following the alternate merging.

Lane changes in dense traffic require a high degree of cooperation between traffic participants because there might be small to no gaps between vehicles for a lane change. For safe lane changes in dense traffic, the lane change assistant has to find an appropriate gap between vehicles on the target lane and position itself next to this gap in order to demand the cooperation of other traffic participants (FR_3). This requirement does not only apply to the merging in front of constriction but also to lane changes between highway lanes in dense traffic and to entering and leaving of highways — especially in weaving areas of interchanges (cf. Section 3.2.1.2).

Lane changes on highways are supposed to improve or sustain the progress of the automated ego vehicle towards its target destination. Therefore, each lane change has to be evaluated under consideration of its benefit for the overall progress of the vehicle's journey (cf. FR_5_1 in Table 3.2). The evaluation of each lane changes has to consider the cost and distance ratio towards the target destination (FR_5_1_1) under consideration of dynamic traffic (FR_5_1_2) and timing restriction from prior events (FR_5_1_3). In front of constrictions, a benefit estimation is not necessary because a lane change is inevitable for the progress.

Table 3.2.: Requirements for the benefit of lane changes.

ID	Description
FR_5	The system must decide if a lane change is beneficial.
FR_5_1	In case, the system is in the automated driving mode, the system has to determine if a lane change is beneficial based on the current traffic situation.
FR_5_1_1	The system shall be able to determine whether a lane change is beneficial based on a cost / distance metric to follow the route towards the navigation destination.
FR_5_1_2	The system shall be able to assess the dynamic traffic situation whether changing the lanes would result in a dynamic benefit.
FR_5_1_3	The system shall be able to assess, whether a lane change is beneficial based on timing restrictions regarding prior driving events.

Lane changes may have a short-term benefit but may result in a disadvantage in the mid or long-term run. For example, overtaking slower vehicles may lead to a short time advantage, but the traffic situation could later prohibit the automated vehicle to take its intended exit ramp. The short-term benefit for the overtaking has been negated by its mid- or long-term disadvantages resulting in an overall disadvantage.
Besides lane changes between highway lanes, the lane change assistant has to cope with lane changes to off-ramps and from on-ramps in order to enter and exit highways. The corresponding requirements for these use cases are presented in the following sections.

3.2.1.2. Entering and Exiting Highways

Besides lane changes between highway lanes, the lane change assistant has to autonomously change from on-ramps to the ongoing highway lanes in order to drive onto highways or change to off-ramps in order to leave highways (cf. FR_6 in Table 3.3). Entering highways differs from lanes changes between highway lanes (cf. Table 3.1) in the way, that limited to none knowledge about the current situation on the highway is available for the lane change assistant. Furthermore, the distance and time available for a lane change to the ongoing lanes are limited by the length of the on-ramps. The lane change assistant has to consider these time and space limitations as well as other traffic participants while entering highways.
The contrary use case to entering highways is the exiting of highways. The lane change assistant must be able to change to off-ramps in order to exit highways. Analog to entering highways, lane changes to off-ramps cannot be postponed indefinitely. The length of off-ramps defines the space and time available for the automated ego vehicle to perform its lane change to the off-ramps. These space and time limitations require the vehicle to swiftly arrange itself with other traffic participants in order to leave the highway in less than 100 m after the off-ramp begins (cf. FR_6_1 in Table 3.3). In dense

Table 3.3.: Requirements for entering or leaving the highway.

ID	Description
FR_6	The system shall be able to perform lane changes to enter a highway on an on-ramp.
FR_6_1	The system shall be able to perform lane changes in onto off-ramps less than 100 m after the off-ramp started.
FR_6_2	The system shall be able to perform lane changes in weaving areas of highway interchanges with a length of less than 200 m.
FR_6_3	The system shall prevent lane changes towards areas with an adjacent highway on-ramp.

traffic, this often requires the cooperation of other traffic participants. For example, the automated ego vehicle drives on the mid lane while a convoy of vehicles drives on the right lane. For the automated vehicle to successfully take the exit ramp, it would have to position itself adjacent to an available gap between the vehicles of the convey and to rely on the cooperation of trailing vehicles in convoy.

Highway interchanges, e.g., cloverleaf interchanges, present a special case for entering resp. leaving highways. In cloverleaves, on-ramps and off-ramps overlap. Traffic leaving the highway has to merge with traffic entering the highways. The lane change assistant has to be able to perform lane change in these weaving areas of highway interchanges with a length less than 200 m (cf. FR_6_2 in Table 3.3). This requirement applies to entering the highway and to exiting the highway (see above). Lane changes in weaving areas require the cooperation of other traffic participants. The automated vehicle must be able to position itself adjacent to an appropriate gap between vehicles on the target lane without interrupting the traffic flow.

Opposite to lane changes between highway lanes, estimations about benefits of lane changes to off-ramp and from on-ramps are not necessary. These lane changes are inevitable in order to proceed with the optimal route towards the final destination as planned by the navigation (cf. Fig. 2.7). In case of entering and leaving highways, lane changes can only vary in certain distance resp. timing windows. These windows correspond to the length of the on- resp. off-ramps (see above).

On-ramps also impact normal lane changes between ongoing lanes of highways. For safety reasons, lane changes towards merging areas for the rightmost lane and on-ramps should be avoided (cf. FR_6_3 in Table 3.3). Lane changes to such merging areas impose the risks of collisions with oncoming traffic. Vehicles entering the highways on on-ramps are forced to change to the ongoing lanes of the highway. Even though entering vehicles have to give priority to vehicles on the ongoing lane of the highway, entering vehicles are less likely to consider vehicle on lanes other than the rightmost highway lane. In the worst case, a collision will be inevitable if a vehicle from these ongoing lanes and an oncoming vehicle simultaneously change towards the merging area on the rightmost lane.

Table 3.4.: Requirements considering traffic participants.

ID	Description
FR_7	The system must decide if a lane change is possible in a given traffic situation.
FR_7_1	The system has to assess the traffic situation based on what is relevant for the decision making.
FR_7_1_1	The system has to evaluate whether a lane change to a neighbor lane is possible based on objects on that neighbor lane behind itself.
FR_7_1_2	The system has to evaluate whether a lane change to a neighbor lane is possible based on objects on that neighbor lane in front of itself.
FR_7_1_3	The system has to evaluate whether a lane change to a neighbor lane is possible based on objects on its current lane in front itself.
FR_7_1_4	The system shall evaluate whether a lane change is possible based on approaching objects on the ego lane behind itself.
FR_7_1_5	The system shall be able to consider objects directly next to it with a relative velocity of less than 5 m/s.
FR_8	The system shall take care of flashing the indicator before initiating a lane change at least 1 s before it starts to impose a lateral displacement.

There exist more requirements which are valid for lane changes between highway lanes as well as for the changes to on- resp. from off-ramps. These functional requirements address the perception of the vehicle's environment and the cancellation of initiated lane changes. These requirements are presented in the following sections.

3.2.1.3. Environment Perception and Interaction

The safety of lane changes significantly depends on the perception of the vehicle's environment and interaction with objects in this environment. This is mandatory for lane change between highway lanes as well as lane changes for entering and leaving highways. The lane change assistant has to consider other traffic participants in its vicinity for the planning and execution of lane changes. Based on traffic participants on the adjacent neighbor lanes and the lane on which the automated ego vehicle is driving the possibility of (safe) lane changes has to be evaluated. No lane change must endanger any other traffic participant.
The Table 3.4 presents requirements which define restrictions for lane changes under consideration of other traffic participants. All restrictions are valid for the previously presented use cases (cf. Sections 3.2.1.1 and 3.2.1.2). Important cases are approaching vehicles from behind and slower vehicles in front of the automated vehicle (FR_7_1_1

Table 3.5.: Requirements considering the road infrastructure.

ID	Description
FR_9	The system has to evaluate whether a lane change is possible based on the current infrastructure.
FR_9_1	The system has to evaluate whether a lane change is possible based on the lane marking type of the appropriate lane boundary.
FR_9_2	The system has to evaluate whether a lane change is possible (necessary) based on the type of the ego and neighbor lanes.

to FR_7_1_4). Faster vehicles behind the automated ego vehicle might drive onto the automated ego vehicle if the automated ego vehicle changes to the lane of the approaching vehicles. Vehicles in front of the automated vehicles on the same lane might impose the possibility that the automated ego vehicle will drive onto these vehicles while changing the lane if the distance to these vehicles is insufficient. Vehicles next to the automated vehicle on either neighbor lane must be perceived even though their relative velocity might be very low (FR_7_1_5).

Traffic participants require automated vehicles to behave plausible and predictable. The lane change assistant has to activate corresponding direction indicators before all lane changes (FR_8) in order to inform other traffic participants about its intention and to give them sufficient time to react.

Additional to traffic participants, the lane change assistant has to consider the highway infrastructure, e.g., road lanes and their markings, in its planning and execution of lane changes (cf. FR_9 in Table 3.5). The lane change assistant must not violate national traffic codes, e.g., by changing lanes in areas where lane markings or signs prohibit lane changes (FR_9_1). Each highway lane commonly has a type assigned which defines if the automated ego vehicle is allowed to drive on this lane resp. change to this lane. For example, emergency lanes, dedicated bus lanes, on- and exit ramps are some particular lane types on which a vehicle is only allowed to drive on in exceptional cases, e.g., emergencies or to leave the highway. The lane change assistant has to detect and evaluate the type of lanes and restrict the possibilities of lane changes accordingly. In case a highway lane does not exist, or its type is not valid, a lane change to this lane is not possible. In case the current driving lane is invalid a lane change has to performed immediately (FR_9_2).

The perception and interaction with the automated vehicle's environment are not the sole requirements which are mandatory for the lane change assistant in all its use cases. The lane change assistant must be able to cancel initiated maneuvers in all situations based on more recent information about the system, vehicle, and its environment. The requirements addressing the cancellation of maneuvers are discussed in the following section.

Table 3.6.: Requirements considering the cancellation of intimated maneuvers.

ID	Description
FR_10	The system shall be able to overturn a previously made decisions based on more recent information.
FR_10_1	The system shall be able to abort flashing the indicators.
FR_10_2	The system shall be able to abort already initiated lane changes.

3.2.1.4. Maneuver Cancellation

Decision made by the lane change assistant cannot be permanent. Made decisions must be reversible because the environment autonomously changes and is not controlled by the lane change assistant. Decisions about lane changes are made based on the perception of the current environmental situation and assumptions about future traffic situations. Decisions about lane changes will have to be reverted based on more recent information if the traffic has been evolved differently than anticipated (cf. FR_9 in Table 3.6). This requirement includes canceling intended lane changes — indicated by flashing of the direction indicator — (FR_10_1) as well as aborting already initiated lane changes (FR_10_2) and returning to a safe position within the origin lane.
Besides the presented functional requirements for the lane change assistant, additional non-functional requirements addressing the quality of the lane change assistant in terms usability, reliability, performance, efficiency, maintainability, compatibility, security, and portability. Corresponding requirements are described in the following sections.

3.2.2. Usability

The general acceptance of the lane change assistant requires the requirements analysis to explicitly consider the interaction between assistant and passengers (cf. Table 3.7). Not only should the driver initiate a lane change on its own — e.g., if the vehicle has a semi-automated driving mode — but also to override any decision made by the lane change assistant and take over the control of the vehicle at any time (UR_2). This requirement especially applies to critical traffic situations which have been misinterpreted by the lane change assistant.
The driver can overrule the lane change assistant and take control of the vehicle by either holding or turning the steering wheel (UR_2_1 and UR_2_2). Level 3 — conditional automation — requires such an intervention in order for the human driver to meet his role as supervisor and safety fallback (cf. Fig. 2.4).
Besides taking over the vehicle control, passengers require to be informed about the decision making by the lane change assistant. A (graphical) HMI has to inform passengers about the system's current situational knowledge and its intentions — especially about the initiation and execution of lane changes (UR_3). Besides the graphical visualization

Table 3.7.: Requirements addressing the interoperability with the passengers.

ID	Description
UR_1	The system must be able to determine if the driver wishes to change the lane.
UR_2	The driver must be able to override the system at all times.
UR_2_1	The driver has to be able to prevent the vehicle from executing a lane change by holding the steering wheel.
UR_2_2	The driver has to be able to execute a lane change by turning the steering wheel.
UR_3	The system has to inform the passengers about its status and intentions at all times.
UR_4	Passengers shall see on a visually appealing GUI if a lane change is about to be executed and hear the indicator flashing sound.

of lane changes, existing HMI elements, e.g., direction indicator signals should be incorporated into the passenger interaction (UR_4).

3.2.3. Reliability

The reliability of the lane change assistant addresses the ability of the lane change assistant to provide its intended functionality correctly in specific environmental situations. Stakeholders, as well as passengers, expect the lane change assistant to perform safe and correct lane changes. Unsafe lane changes which impose danger for its passenger and other traffic participants have to be limited to a sufficient minimum. Safety standards, e.g., the safety standard ISO 26262 [Int09b], define thresholds for the rate of errors which systems have to meet in order to be judged as safe. For the lane change assistant (cf. Table 3.8), the rate of false estimation of the possibility of safe lane changes has to be less than 1.0×10^{-9} times per hour of driving time (RR_1). Furthermore, the benefit estimation of lane changes must not exceed 2.0×10^{-8} false estimations in one hour of driving time (RR_2).
The environment measurements of sensors are subject to uncertainty. Following the requirements of Table 3.8, the lane change assistant has to cope with temporarily incorrect or uncertain measurement data and still be able to process safe lane changes (RR_3). Uncertainties have to be considered for all measured values about objects in the automated ego vehicle's vicinity, e.g., position and velocities of other vehicles (RR_3_1). Furthermore, the rate of incorrect data and their continuous duration should be monitored and evaluated (RR_3_2).
Long durations with incorrect measurement values from sensors may prohibit the processing of safe lane changes because the current belief about the environment does not match the real environmental situation. The last available accurate measurements of the sensors which the lane change assistant could use to mitigate the inaccurate measurements are

Table 3.8.: Requirements addressing the reliability.

ID	Description
RR_1	The system must decide if a lane change in a given traffic situation is possible with an error rate of less than 1.0×10^{-9} errors per hour of driving time.
RR_2	The system must decide if a lane change is beneficial based on the current traffic situation with an error rate of less than 2.0×10^{-8} errors per hour of driving time.
RR_3	The system has to be able to handle uncertain and/or temporarily incorrect measurement data.
RR_3_1	The system shall be able to consider measurement uncertainties of object positions, velocities, accelerations and their existence on a particular lane.
RR_3_2	A metric shall quantify temporary incorrect measurements and cover the duration of the deviation from the correct value.
RR_3_3	The system may not base its current decision on sensor data older than 100 ms if newer data is available
RR_4	The system shall provide consistent driving orders. It shall overturn less than 3.0×10^{-7} decisions per hour of driving time.
RR_4_1	The system shall abort flashing the direction indicators less than 2.0×10^{-7} times per hour of driving time.
RR_4_2	The system shall abort a initiated lane change less than 1.0×10^{-7} times per hour of driving time.

too old. The lane change assistant has to use the most recent data for its processing of lane changes in order to process lane changes appropriate for the current environment situation. Measurements older than 100 ms must not be incorporated into the processing of lane changes if more recent data is available (RR_3_3).

Even though the lane change assistant shall be able to cancel already initiated maneuvers based on more recent information about its state and the state of its environment, these cancellations should not occur too frequently. The overall behavior of the lane change assistant has to be consistent (cf. in Table 3.8). Frequent cancellation of maneuvers indicates that the implementation of the lane change assistant does not sufficiently incorporate the possible future behavior of other traffic participants in the planning of lane changes. The lane change assistant shall not overturn previous decisions more than 3.0×10^{-7} times per hour of driving time (RR_4). Flashing of the direction indicators shall be canceled statistically less than 2.5×10^{-7} times per hour of driving time (RR_4_1) while initiated lane changes should be reverted less than 1.0×10^{-7} times per hour of driving time (RR_4_2).

Table 3.9.: Development Requirements.

ID	Description
DR_1	The system may not use more than 10 % of the computation time of a quad-core desktop computer.
DR_2	The system may not use more than 4 GB of RAM if executed on a desktop computer.
DR_3	A developer shall be able to understand and contribute the lane change module in less than one year.
DR_3_1	Each class and each function shall have documentation in the productive code.
DR_4	A developer shall be able to log the state of the system, the system input data and the system outputs.

The reliability and safety are significant factors in the development of automotive systems because they are related to product liability (cf. ProdHaftG §1, BGB §823 I, BGB §433). An unreliable system causing accidents with casualties or even fatalities may result in high recourse claims against its producer. Therefore, additional analysis is performed to estimate, enhance, and maintain the reliability and safety of such systems. For the lane change assistant, such a safety analysis is presented in Section 3.4.
Requirements which are mainly related to the development of the lane change assistant are present in the following section.

3.2.4. Development

The Table 3.9 presents requirements particular related for the development of the lane change assistant. We aggregate here the requirements from the categories efficiency, maintainability, compatibility, and portability from the quality standard ISO-IEC 25010:2011 [Int11b]. The provided system specification for the lane change assistant does not extensively address these categories in order to address each category on its own.
As minimal performance requirements for the system development, the implementation of lane change assistant must not use more than 10 % of computation time available on a quad-core desktop computer (DR_1) and less then 4 GB of random access memory (RAM) (DR_2). These requirements address the efficiency and portability as the system must not only operate in the automated ego vehicle but also be executed and simulated in the development on common desktop computers.
The maintainability of the lane change assistant requires novel developers to understand the implementation of systems efficiently. They only should require at most one year to understand the internal structure and functionality of the lane change assistant (DR_3). Therefore, the functionality of each class and function has to be sufficiently documented in the productive code (DR_3_1). Furthermore, developers must be able to log the system inputs, state, and outputs at runtime in tests and productive environments in order to understand the system behavior and identify faulty system behavior (DR_4).

Security has not been addressed by any requirement for the lane change assistant.
From the set of requirements, an initial concept of the lane change assistant is designed. This system design is described in the following section.

3.3. System Design

The requirements gathered in the requirements analysis (cf. Section 3.2) represent the exterior functionality of the system. The system design addresses the internal structure of the system — how the system functionality is separated into smaller functional components and hardware components. A decomposition into system parts with clearly defined interfaces serves the implementation of the system in multiple ways; system parts can be developed, modified, extended, and tested independently from other system parts.
From the system specification of the lane change assistant (see the previous section), a concept of the system is designed (cf. Fig. 3.2). The concept incorporates functional and technical architectures. The functional architecture partitions the system into a hierarchy of functional components. It is described in more detail in Section 3.3.1. The technical architecture, which is described in Section 3.3.2, defines a set of hardware components and their connections within the vehicle.

3.3.1. Functional Architecture

In the functional architecture, the system is partitioned into a hierarchy of functional components. The top hierarchy level represents the lane change assistant in its entirety. Each functional component can be decomposed into a set of subordinated functional components from which each implements a part of the functionality of the superordinate component (cf. Fig. 3.3). Inputs of the superordinate components are processed by the set of subordinate components resulting in outputs of the superordinate components. For all leaves of the component hierarchy — the *atomic functional components* — algorithms, communication, and data concepts are defined and documented by corresponding component specifications.
Components are self-contained units which only interact with their environments — other components — via defined interfaces (cf. Definition 3.4). Interfaces define the functionality which components offer their environments — other components — by defining the data that can be exchanged via these interfaces. Each functional component can be implemented and tested against its interfaces. This partition enables the implementation and verification of each functional components independently from other components of the same hierarchy level. Components can be used in different contexts (environments) if the interfaces remain consistent.

Definition 3.4 (Component)**.** A [software] component is a unit of composition with contractually specified interfaces and explicit context dependencies only *[CDS02].*

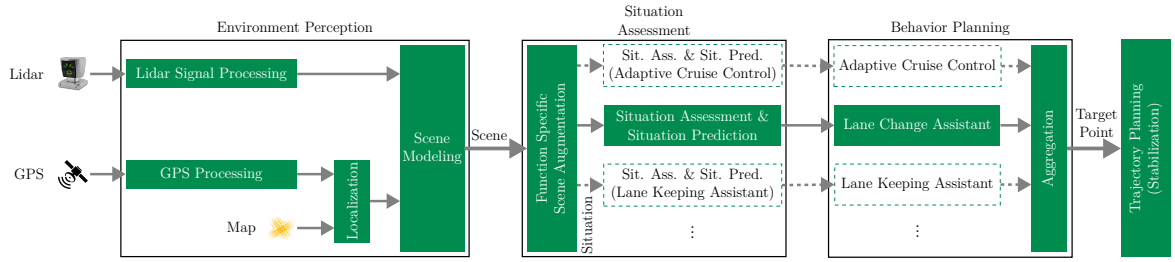


Figure 3.3.: Functional architecture for the processing of lane changes (level 1).

At hierarchy level 1 of the functional architecture, the lane change assistant consists of three functional components — *environment perception*, *situation assessment*, and *behavior planning*. This processing chain from sensors to the trajectory planning (stabilization) is presented in Fig. 3.3.
The environment perception address the perception of the vehicle's environment by sensors, the processing of sensor data, and the generation of a comprehensive representation of the vehicle's current environment — the *scene* (cf. Definition 2.25). This scene is assessed by the *situation assessment* to provide the planning of lane changes with a minimal but sufficient environment description — the *situation* (cf. Definition 2.9). A detailed explanation and distinction of the terms *scene* and *situation* can be found in [Ulb+15].
Possibilities and benefits of lane changes in the current situation are evaluated by the *behavior planning* of the lane change assistant. The behavior planning of lane changes is performed on tactical level (guidance) parallel to the behavior of other ADAS, e.g., ACC or LKAS (cf. Fig. 3.3). Each function outputs a corresponding target point. All target points are compared and evaluated. As result, one final target point is elected and passed to the stabilization level (cf. Fig. 2.3). From the final target point, the trajectory planning processes the new trajectory for the vehicle (cf. Fig. 3.3). Each high-level component is described in the following sections.

3.3.1.1. Environment Perception

The behavior planning of lane changes requires a representation of the current state of the vehicle's environment. This environment state has to include the *scenery* — the static environment and objects, e.g., roads and buildings — as well as dynamic objects in the vehicle's vicinity, e.g., vehicles and pedestrians (cf. Section 2.2.1.1). The lane change assistant has to perceive its environment with sensors because the system does not control the environment. The perception component (cf. Fig. 3.3) incorporates a LIDAR sensor to perceive the real environment (cf. [GG14]). There exist LIDAR sensors which provide 360° views around the automated ego vehicle with multiple scanning distances (cf. [Sch10]). The physical signals from the LIDAR sensor, the reflection points of emitted laser beams, are processed into a comprehensive representation of the vehicle's environment — the *scene* (cf. Definition 2.25).
Reflection points of emitted laser beams are preprocessed by transforming these physical signals into a hypothesis about objects in the vehicle's environment. Furthermore, global

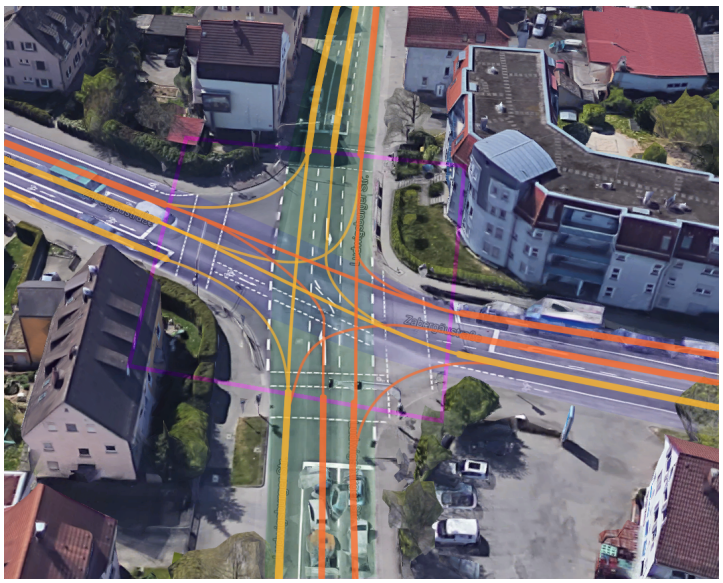


Figure 3.4.: Representation of a road junction as road graph[1].

positioning system (GPS) signals and high accurate road maps are incorporated by the *localization* into the perception in order to enable the *self-estimations* of the vehicle's position and pose in the world. The module *scene modeling* models a comprehensive representation of the vehicle's environment based on the list of objects from the LIDAR sensor and the self-estimation of the automated ego vehicle. The self-estimation allows positioning all detected objects in relation to the automated ego vehicle. The resulting environment model is commonly referred to as *scene* (cf. [Ulb+15]). It serves as the interface between the environment perception and the situation assessment.

3.3.1.1.1. Data Structure of the Scene

The *scene* represents the configuration of the vehicle's environment as an spatial-temporal arrangement from an observers point of view for one particular time stamp — including the *scenery*, dynamic objects, and self representation (cf. Definition 2.25). The scene continuously evolves resulting in a sequence of scenes. Each object in the scene is defined by its type and a set of corresponding properties, e.g., the width, length, velocity, and acceleration of a vehicle. Additional, relationships between environment objects are documented in the scene. For example, vehicles are related to the lanes of the road on which they drive.
The *scene* integrates information about the road network extracted from digital maps with information about the vehicle's current environment provided by *exteroceptive* sensors (cf. Fig. 3.3). Maps describe the static scenery of the road network while the sensors provide up-to-date information about the current traffic situation including dynamic objects and changes of road elements which have not yet been documented in maps. As shown in Fig. 3.5, normal roads consist of two ways with an arbitrary set of lanes for each way.

[1]Scenery taken from Google Maps on 10.4.2017

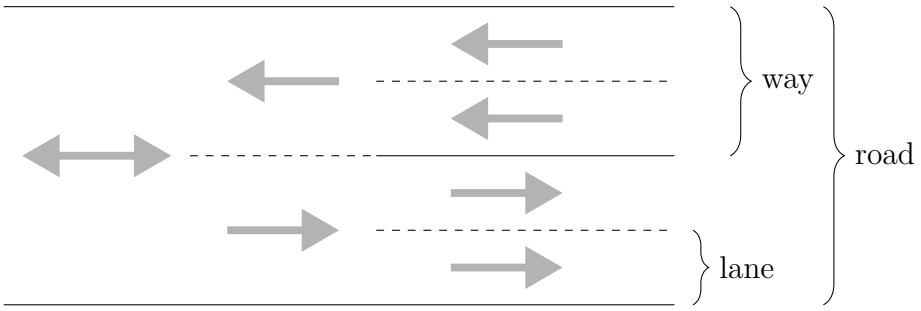


Figure 3.5.: Relationship between road, way, and lane.

Rural roads mostly have one lane per way while highways have two or more lanes per way.
The road network in the *scene* is implemented by a directed *road graph*. The *road graph* include information about road lanes, road intersection and static road elements, like traffic sings, speed limits, or traffic lights (cf. [KH10]). As shown in Fig. 3.4, the real roads and junctions are defined by their set of lanes in the *road graph*. Lanes are represented in Fig. 3.4 as yellow and orange lines representing the two driving directions. Ongoing lanes of each direction are drawn bold. On multi-lane roads, e.g., highways, multiple ongoing lines would exist side by side for each direction. All remaining lines in Fig. 3.4 represent the possible turns which vehicle may take within the junction. The benefit of describing the roads and junctions on lane level is the possibility to model permitted directions in intersections explicitly.
In the *road graph*, road lanes are represented by lane segments and lane segment connections (cf. Fig. 3.6 [Ulb+14]). The lane segment connectors correspond to vertices in a directed graph while the lane segments represent the graph's edges. Lane segments subdivide each road lane into smaller parts allowing to address variations of lane properties, e.g., markings or curvature. Lane segments start and end at lane segment connectors. The direction of lane segment is inherently defined by its direction in the graph and corresponds with the driving direction of vehicles on the corresponding real lane. Each lane segment commonly has a left and right lane boundary which represent the corresponding lane markings. Adjacent lane segments share one common lane boundary. Lanes and roads are composed of sequences of lane segment connectors and sets of adjacent lane segments. The graph-based model can not only be used for the guidance but also the navigation of the automated ego vehicle (cf. Section 2.1.4.1).
In the data structure for the scene, the road network (cf. Fig. 3.7) is represented by the classes *lane segment*, *lane segment connectors*, and *intersections*. Each lane segment has a set of attributes to describe its properties. The identifier *id* of each lane segment is a concatenation of the identifiers for the corresponding road, way, and the number of the lane (cf. Fig. 3.5). This concatenation enables to explicitly identify a lane segment in the road graph. Each lane segment has a type which defines if vehicles are allowed to drive on this lane segment. Types of lane segments range from *NormalLane*, *ForbiddenLane* to *EmergencyLane*. The position of each lane segment in the world is defined by its start (*startPosition*) and end points (*endPosition*). Any position of objects in the scene are

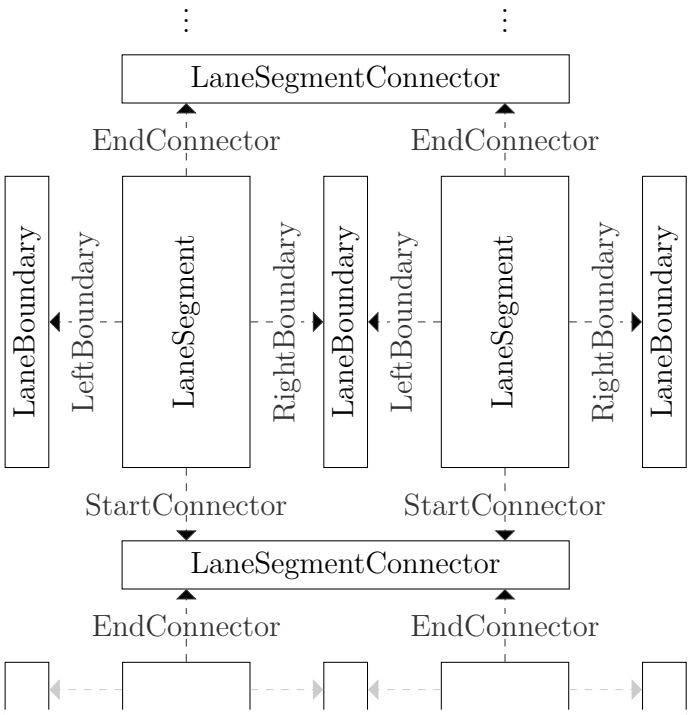


Figure 3.6.: Modeling of lanes segments, connectors, and boundaries.

defined in the world-centered coordinate system — universal transverse Mercator (UTM) (cf. [Chr02]). The class *LaneBoundary* represents the left and right boundaries of each lane. Each lane boundary has a type that corresponds to its lane marking. Possible types for the lane boundary are *solid*, *dashed*, and *unknown*. Further properties address the *width*, *curvature*, and the envisaged *offset* of vehicles on this lane. [Ulb+14]

The data element *intersection* models all permitted driving directions within road intersections. Besides ongoing lanes within the intersection (bold lines in Fig. 3.4), additional edges model turning lanes by connecting lane segment connectors of approaching lanes to connectors of leaving lanes within the intersection (thin lines in Fig. 3.4). One incoming lane segment may connect to several outgoing lane segments. Each extra edge represents one turning lane with its corresponding rule of priority — e.g., left-before-right or traffic light controlled. These edges solely define the possibility of turning but no concrete path which vehicles have to follow while turning. The automated ego vehicle itself has to determine its trajectory on these turning lanes without explicit paths under consideration of national traffic codes.

However, large intersections may require the explicit modeling of turning lanes and their paths. The model of lane segments and lane boundaries from Fig. 3.6 can be reused to describe turning lanes and their characteristics, e.g., curvature and length. The lane segments and boundaries enable to model the correct driving corridor for turning lanes within intersections. Turning lanes with varying curvatures can be modeled by multiple

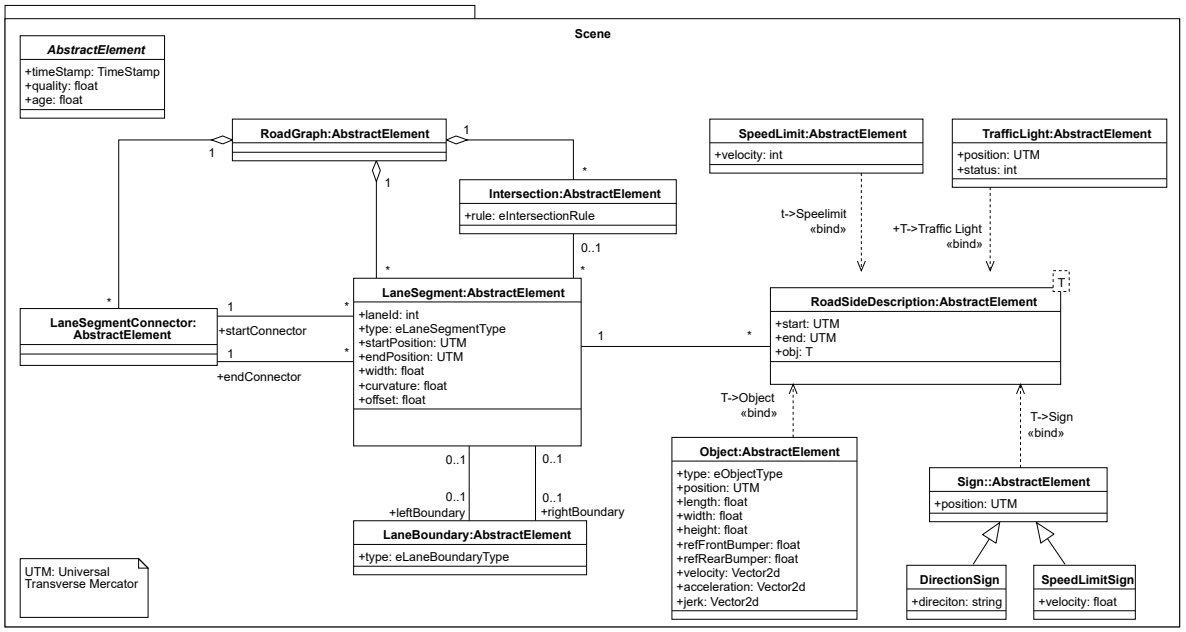


Figure 3.7.: UML class diagram of the Scene.

connected lane segments with individual but constant curvatures. The connections of these individual lane segments incorporate lane segment connectors.

Objects, e.g., signs, vehicles, pedestrians, traffic lights, perceived by the vehicle's sensors have their position in world coordinates (UTM) but are also matched to corresponding lanes segments by the template class *RoadSideDescription* (cf. [KH10]). For example, the matching of signs and vehicles to lane segment will increase the efficiency of the situation processing (cf. Section 3.3.1.2). The positions of objects on lane segments are defined by a *start* and *end* point. The speed limit (cf. Fig. 3.7) is the only object in the *scene* that has no position attribute. The speed limit defines a sector with limited velocity. This sector can be defined based on the start and end point of the class *RoadSideDescription*. Every class in the scene can have its own set of attributes. For example, traffic lights can have their status as attributes — e.g., green, red or yellow — while traffic sings have parameters based on their type. A direction sign will have a description while a speed limit sign has the maximum velocity as an attribute. The set of objects displayed in Fig. 3.7 for the scene is an example and does not intended to be complete. Additional object types can be defined and added to the *scene*, e.g., parking prohibition zones or stop signs at intersections.

Dynamic objects, e.g., vehicles, trucks, or pedestrians, are represented in the scene by the class *Object*. The type of an object is defined by the enumeration *eObjectType* and includes types for e.g., vehicles — including the automated vehicle itself — or pedestrians (cf. Fig. 3.7). The position of each object is given in UTM and matched to the corresponding lane by the class *RoadSideDescription*. Opposed to static objects, e.g., signs or traffic lights, dynamic objects autonomously change their position and orientation. The movement of dynamic objects is described by the attributes *velocity*,

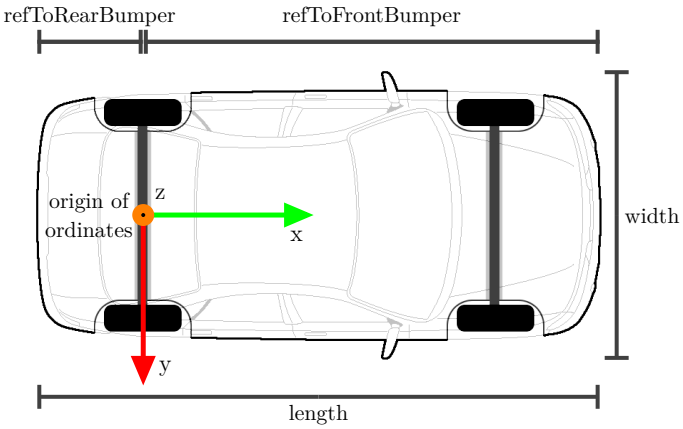


Figure 3.8.: Coordinate system, reference points, and measurements for vehicle positioning.

acceleration , and *jerk*. Each attribute is direction dependent and defined as a vector. As shown in Fig. 3.8, the size of an object is denoted by its *length*, *width*, and *height*, while the parameters *refToFrontBumper* and *refToRearBumper* define the distance of the front resp. rear bumper to vehicle's origin of ordinates. The origin of ordinates for vehicles commonly resides in the center of the rear axle. Distances between environment objects are measured between their respective origins of ordinates. Objects, e.g., pedestrians or bicycles, may be modeled as distinctive subtypes with more specific attributes.

All elements of the scene inherit from the class *AbstractElement*. The class *AbstractElement* defines the attributes *timeStamp*, *quality*, and *age*. The attributes quantify the adequacy of scene elements for representing their corresponding real-world objects because objects' parameters in the *scene* are subject to uncertainty. The uncertainty of parameters resides from the uncertainty in the perception of real-world objects by the vehicle sensors. The attribute *quality* represents the confidence of the perception to have correctly captured the real world object and its properties. The attribute *timeStamp* addresses the age of scene elements with respect to its processing in the lane change assistant by storing the time of the element's last update. The attribute *age* addresses the confidence for the element to correctly represent its real-world object based on the time the element has been known. These attributes enable to reasoning about the validity of stored information in the scene for the current real-world situation. New objects, objects with lagging updates, or objects with low quality might not sufficiently correlate to their corresponding real-world objects and have large deviations for the attribute values. In the worst case, such parameter deviations might lead to collisions with other traffic participants. Subsequent modules in the processing chain of the lane change assistant incorporate this adequacy information about the *scene* in their decision making.

The *scene* holds a vast scope about the automated ego vehicle's environment which might not be required in full extension for the lane change assistant. Processing this unnecessary data for each function would be inefficient. Objects and information with no impact on the decision making of functions would have to be processed. For example, an ACC does

not require the processing of information about the traffic next and behind the automated ego vehicle. Therefore, the scene is augmented by the situation assessment in order to reduce the complexity of the environment representation to a level which is sufficient for the behavior planning by the lane change assistant to process. The augmentation of the scene by the *situation assessment* is described in the following section.

3.3.1.2. Situation Assessment

The situations assessment consists of two processing stages (cf. Fig. 3.3). First, the goal- and value-independent scenes from the perception are transformed into function-specific representations for each ADAS function. Afterward, these function-specific representations are further assessed and augmented for the *behavior planning*. The assessment and augmentation for each function includes the estimation of relevant parameters that are not directly measured within the scene or are subject to uncertainty.

3.3.1.2.1. Scene Augmentation

The component for the *function specific scene augmentation* augments goal- and value-independent scenes with function-specific and permanents goals of the vehicle and its functions (cf. component *function specific scene augmentation* in Fig. 3.3). The result is a function specific representation of the environment and vehicle itself. This specific representation is denoted as *situation* and is specifically tailored for each particular function. A *situation* encapsulates all necessary but sufficient information for the behavior planning for one particular ADAS function (cf. Definition 2.9). Relevant elements of the scene may be augmented with additional symbolic information resp. actions aspects related to the particular function while irrelevant elements are omitted from the *situation* (cf. [PD02; Sch11b]). This approach limits the complexity of the environment description to a level that is necessary but sufficient for corresponding ADAS functions to process their actions. The same scene may evolve into different situations based on the information requirements of each function [Ulb+15].

As the scene encapsulates more information than lane change assistant requires for its decision making, irrelevant scene elements are omitted from situations for the lane change assistant (cf. Fig. 3.9). The lane change assistant requires only information of, e.g., position and type of the ego (same lane as the automated ego vehicle) and both adjacent neighbor lanes as well as positions and movements of dynamic objects, e.g., other vehicles, driving on these lanes inside the sensor ranges. For example, vehicles driving on distant lanes are excluded from the situation because they cannot intervene with the lane changes of the automated ego vehicle. Scenery elements other than the road and its lanes, e.g., trees or house, are not transferred into the situation. Even road signs may be excluded from the situation if they are irrelevant for the lane change planning. Signs forbidding lane changes are included in the situation while signs for priority rules or direction signs are excluded (cf. Fig. 3.9).

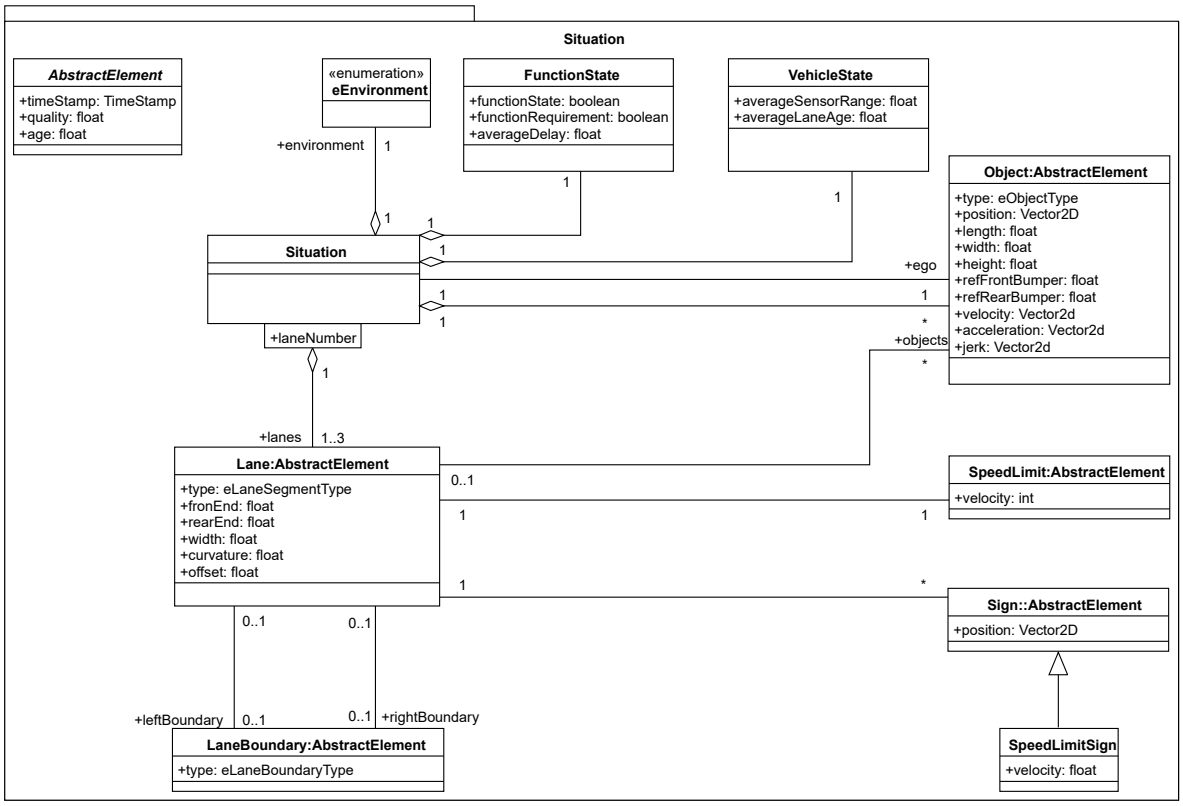


Figure 3.9.: UML class diagram of the situation.

3.3.1.2.2. Data Structure of the Situation

Figure 3.9 displays the data structure of the *situation* as a unified modeling language (UML) class diagram (cf. [OB12]). The situation is a partial mapping of the *scene* with a narrow scope. Elements of the data structure for the *scene* are adopted for the data structure of the *situation* (cf. Fig. 3.9). These elements are the lane segments, the lane boundaries and relevant objects for the lane change assistant, e.g., vehicles or signs. Other road graph elements, e.g., the lane segment connector and intersection, are excluded from situations. With the transformation from *scene* to *situation*, the reference point for the positioning of all objects changes from a world-coordinate system (UTM) to positioning relative to the automated ego vehicle. In the situation, the positions of all objects are stored as vectors relative to the origin of ordinates of the automated ego vehicle (cf. Fig. 3.8).

Situations encapsulate all objects which are necessary for the representation of system state and environment state at a certain time stamp (cf. Fig. 3.9). A situation contains information about the automated *ego* vehicle (cf. Definition 3.5), a list of all vehicles in the vicinity of the ego vehicle — *objects* —, a list of road lanes — *lanes*. Compared to the scene (cf. Fig. 3.7), objects are directly associated to their corresponding lanes and not via the template class *RoadSideDescription* (cf. Fig. 3.9). All object properties stored in the situation still posses the uncertainty introduced by the imperfect vehicle

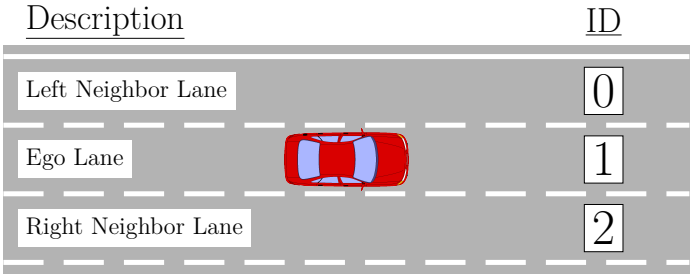


Figure 3.10.: Lane number in the scene.

sensors. The reader is referred to [Ulb+15] for more detailed descriptions of terms *scene* and *situation*.

Definition 3.5 (Ego Vehicle)**.** *The term ego vehicle denotes the automated ego vehicle which is (partially) controlled by the considered autonomous vehicle system.*

In the *situation* for the lane change assistant, the road is represented as a set of three lanes (cf. Fig. 3.7). Each lane is defined by the class *Lane*. For the lane change assistant, the lane on which the automated ego vehicle is driving and its left and right neighbor lanes are of interest (cf. Fig. 3.10). All other lanes are omitted in the *situation* for the lane change assistant. Each of the three lanes has a fixed *lane number*. Each lane can be directly referenced by its respective *lane number*. The lane numbers are assigned to the three lanes in ascending order starting with the left lane. The ego lane always has the lane number one. Therefore, the labeling of each lane does change with lane changes. In case the automated vehicle performs a lane change to its left neighbor lane, the ego lane of the initial *situation* before the lane change becomes the right neighbor lane and obtains the lane number 2 while the left target lane of the lane changes becomes the new ego lane with lane number 1 in the *situation* after the lane change. Each lane has a *type* which will define if vehicles are allowed to drive on this lane. The possible lane types correspond to the types used for lane segments in the *scene*. All three lanes — the ego and the two adjacent neighbor lanes (cf. Fig. 3.10) — are always present in the situation even though a lane might not exist — e.g., on two-lane highways. In case a lane does not exist, the type of this lane is *non-existing*.
The lane parameters *width*, *curvature*, and *offset* are adopted from the *scene* (cf. Fig. 3.7). Lane markings are of additional importance for lane changes because the markings can constraint the lane changes. For example, a solid lane marking prohibit a lane change past this lane marking. Each lane in the *situation* has a left (*leftBoundary*) and a right (*rightBoundary*) marking. These markings corresponds the lane boundaries of the *scene* (cf. Fig. 3.7). The type of each marking is defined by the enumeration *eBoundaryType* and can be one of three types; *solid*, *dashed*, or *unknown*. The type *unknown* is used for missing, unrecognized, not clearly visible, or unknown patterns of lane markings. Opposed to the *scene*, a lane does not have a start and end position in world-system coordinates but two attributes *frontEnd* and *rearEnd*. The attributes *frontEnd* and

rearEnd define the length of each lane in relation to the position of the automated ego vehicle. The lengths of lanes are limited by the maximum ranges of the vehicle sensors. Elements from the scene which are relevant for planning lane changes, e.g., objects, signs, and speed limits, are adopted in the *situation*. Relevant elements are directly associated to their corresponding lanes and not via the *RoadSideDescription* as used in the *scene* (cf. Fig. 3.7). A special case is the speed limit because only one possible speed limit can be associated with each lane in the *situation*. Only the speed limit at the current longitudinal position of the automated ego vehicle is considered for each lane. All Elements in the *situation* which have been adopted from the *scene* still have the information about their time of the last update, their age, and their quality introduced by the class *AbstractElement* (cf. Fig. 3.9).
In addition to the road and its lanes, the lane change planning depends on objects in the automated vehicle's vicinity. These objects are represented by the class *Object* in the *situation* (cf. Fig. 3.9). The *objects* in the *scene* adopt the information about *position*, dimensions, *velocity*, *acceleration*, and *jerk* amongst other parameters of the dynamic objects from the *scene* (cf. Fig. 3.7). Values for the attributes *position* are transformed from the world-coordinate system (UTM) in the *scene* into the coordinate system relative to the automated vehicle for the *situation*. Each object is assigned to one of the three existing lanes in the *situation* — the ego and each adjacent neighbor lane. Objects on any other lane than the ego or two adjacent neighbor lanes are omitted from the *situation*. The assignments to lanes and the dimension parameters of objects enable the lane change assistant to estimate the free driving and occupied areas of the road. The different types of objects are defined by the attribute type and its enumeration *eObjectType* similar to the *scene*. One of the possible types is the type *vehicle*. In the *situation*, the class *object* with the type *vehicle* is identically used to define the automated ego vehicle as well as other vehicles in its vicinity.
The situation can be seen as the compilation of relevant scene data and its interpretation from a function-centric perspective (cf.[Rei+10]). Besides encapsulating relevant elements from the *scene*, the *situation assessment* augments elements with additional symbolic information resp. situational and functional aspects. This augmentation enables the *behavior planning* to process appropriate maneuvers for the automated ego vehicle in the current traffic situation. In the *situation* (cf. Fig. 3.9), special classes are introduced for the representation of information about the vehicle state (*VehicleState*) and the modules of the environment perception (*FunctionState*). Furthermore, the enumeration *eEnvironment* reflects the current road domain in which the automated vehicle is currently driving; possible values are *highway*, *rural*, and *urban*. The domain is derived from information about the configuration of lanes, occurrence of special signs, and speed limits in the *situation*. For example, highway signs directly indicate the start of the highway domain whereas speed limits of less than $50\,\frac{\mathrm{km}}{\mathrm{h}}$ are more likely to reflect urban areas. The information about the current domain is necessary in order to restrict the activation of the lane change assistant to highways.
The class *FunctionState* addresses the requirements which ADAS functions impose on the correct processing by the environment perception. Each function defines its requirements for the correct processing of perception modules, e.g., *localization* and *scene modeling*.

These requirements are stored in an array with fixed positions for each module in the attribute *functionRequirement*. The actual state of each perception module is defined in the attribute *functionState*. A comparison of the attributes in *functionRequirement* and *functionState* will determine for a time stamp if all requirements of an ADAS function are met by the environment perception and the data stored in corresponding *situations*. Based on the time stamps of all objects from the *scene*, the average delay of elements in the situation since their last update can be calculated and estimated. The average delay enables to evaluate if a situation still corresponds to the current real world. The average delay of situation elements is stored in the attribute *averageDelay* of *FunctionState*.

The class *VehicleState* is introduced to the *situation* in order to address the capabilities and restriction of the vehicle and its hardware systems, e.g., sensors. *VehicleState* encapsulates information about the quality of the perception hardware system including the current average ranges of sensors and the average time which environment objects have been tracked without any loss or interruption. The average sensor range is defined in the attribute *averageSensorRange*. Information about the sensor ranges is necessary for the estimations about the correctness and safety of the *behavior planning*. In case, the sensor ranges are limited due to, e.g., fog or snow, the vehicle might be unable to provide sufficient information about critical environment objects, e.g., hidden vehicles. These objects could potentially interfere with lane changes of the automated ego vehicle. In the worst case, the automated ego vehicle performs an unsafe lane change resulting in a collision with fatalities. Furthermore, sensors can break or be covered by, e.g., dirt, resulting in faulty sensor data which must be ignored by ADAS functions for their *behavior planning*. The attribute *averageAge* aggregates the age of relevant objects, e.g., vehicles, signs, and lanes (cf. class *AbstractElement*) in order to efficiently reason about the consistency of their *object tracking*.

3.3.1.2.3. Situation Assessment and Situation Prediction

The planning of lane changes requires the assessment of the vehicle and its environment as one comprehensive state — in following denoted as *system state*. Some aspects of this system state can be directly observed from *situations*, e.g., distances and velocities of surrounding vehicles, while other aspects have to be estimated or derived from these observations. Unobservable aspects are commonly modeled as hidden (state) variables (cf. [UM15a]). As the measurements included in the situations, the assessment of the system state is subject to uncertainty that has been introduced by the sensor perception. The component *situation assessment and situation prediction* transfers measurement information, e.g., distances and velocities of vehicles, from the *situation* into an aggregated belief about the *system state*. The *belief* about a system state corresponds to the probability that a specific configuration of the situation is valid for the current time step.

Definition 3.6 (System State)**.** *The* system state *encompasses the internal state of the autonomous vehicle system and the state of the system's environment. The internal state of autonomous vehicle systems is represented by the current values of its internal parameters, and the state of the system environment is defined by the positional and behavioral parameters of static and dynamic objects in the vicinity of the autonomous vehicle systems.*

For the planning of lane changes, Ulbrich and Maurer introduce two high-level parameters which estimate the possibility and benefit for changes to the left resp. right neighbor lane [UM15a]. The two parameters are independently evaluated for lane changes to the left neighbor lane and lane changes to the right neighbor lane. Following the system's requirements (cf. Section 3.2), additional hidden parameters have to be calculated in order to estimate these high level parameters [UM15a]:

Lane Change Possible Estimation: The estimation about the possibility of lane changes has to consider the dynamic traffic situations, the infrastructure, and restrictions by the system's abilities (cf. the requirements in Section 3.2). Based on the relative position of each object, objects have to identified that might interfere with a lane change.

Besides the dynamic traffic also the infrastructure has to allow for lane changes. The type of the Lane (cf. *eLaneType* in Fig. 3.7) must not restrict the automated vehicle from driving on this lane e.g., emergency lanes. Additionally the corresponding lane marking — the left marking for a lane change to the left neighbor lane (cf. *leftMarking* in Fig. 3.7) and the right lane marking for a lane change to the right neighbor lane (cf. *rightMarking* Fig. 3.7) must allow lane changes to these lanes, e.g., the marking type *dashed*.

In dense traffic, lane changes might require the automated ego vehicle to position itself adjacent to the most appropriate gap between vehicles on the target lane. Therefore, positions of surrounding vehicles are compared with each other in order to estimate gaps between them. The automated ego vehicle has to position itself adjacent to an appropriate gap before its lane change. This relates to the requirement FR_3 in Section 3.2.1.1.

Furthermore, permanent and temporal limitations of the vehicle, e.g., limited perception ranges of sensors, have to be considered. Vehicles and the road itself might obstruct the viewing range of sensors and prohibit the detection of vehicles which could interfere with a safe lane change.[UM15a]

Lane Change Beneficial Estimation Advantages of lane changes have to be evaluated based on the potential relative velocity gains in different regions around the ego vehicle. The flow of the traffic and the resulting velocity and time gains have to be estimated for each lane. A lane change to a neighbor lane which traffic flow is overall slower might not be more beneficial as remaining on the current lane behind a slow vehicle.[UM15a]

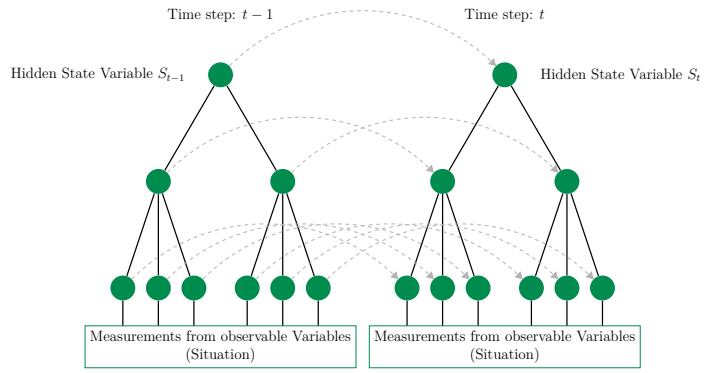


Figure 3.11.: Graph of a Dynamic Bayesian Network.

> All beneficial estimations have to be performed under consideration of long-time disadvantages resulting from immediate behavior changes. Lane changes might be beneficial in all situations even though they might be possible. For example, if the automated ego vehicle passes a slower vehicle only to later miss its intended highway exit due to dense traffic, the initial benefit of the lane change will be mitigated and results in an overall disadvantage. Such a lane change would neglect the route processed by modules of the navigation layer (cf. Fig. 2.7) and result in a less efficient alternative route. [UM15a]

All estimations about the possibility and benefit of lane changes have to consider the inherent uncertainty in the *situation* explicitly (cf. Section 3.3.1.1). The processing chain of the lane change assistant might accumulate uncertainties and deviations in its data leading to potential misinterpretation of the current environmental situation. In the worst case, improper lane changes for the current real-world situation might threaten the life of the vehicle's passengers, other vehicles, and their passenger.
Bayesian networks [FGG97] are able to explicitly consider the uncertainty of situation data in the estimation of the system state (cf. [UM15a]). While some state parameters are directly measurable, e.g., the vehicle's velocity, other parameters are hidden. These non-measurable state parameters are described as hidden state variables by the Bayesian networks. Hidden state variables are estimated from the other hidden state variables and measurements by conditional probability. Bayesian networks can be represented as directed acyclic graph where each node represents a hidden state variable and edges between these nodes depict their dependencies (cf. Fig. 3.11).[UM15a]
The *situation assessment and situation prediction* for the lane change assistant uses a dynamic Bayesian network for the processing of an aggregated belief about the system state (cf. [DGH92]). A dynamic Bayesian network extends a regular Bayesian network by introducing temporal dependencies among nodes (cf. Fig. 3.11). Arbitrary many previous values of a hidden state variable can be incorporated into the estimation of this variable for the current time step. For the lane change planning, the last value of a variable is considered. The value of state variable $\mathcal{S}_t$ at time step t is estimated based on the value of that particular state variable $\mathcal{S}_{t-1}$ at the previous time step $t-1$

and the latest measurements $\mathcal{V}_t = \{V_t(O_i) \mid O_i \in \mathcal{O}, 1 < i < n\}$ of the observable state variables $\mathcal{O} = \{O_1, \ldots, O_n\}$, e.g., object distances (cf. Fig. 3.11 [UM15a]). The result of the dynamic Bayesian network for the lane change assistant is the aggregated belief about the current system state. [UM15a]
Measurements of the lane change assistant system include continuous (e.g., the vehicle velocity) and discrete parameters (e.g., the possibility of a lane change). Dynamic Bayesian networks usually consider only discrete random variables. For the lane change assistant, continuous and discrete state variables have to be considered. It is not possible to explicitly define the conditional probability of dependencies to other variables for continuous state variables. Continuous state variables have to be represented by finite values ranges under the usage of a probability density function (cf. [Par62]). A sigmoid function and the cumulative distribution function of the normal distribution are used for the lane change assistant.[UM15a]

3.3.1.3. Behavior Planning

The *behavior planning* uses the believes about system states from the *situation assessment* to derive behavior decisions (cf. Fig. 3.3). Under the complexity of the real world and the uncertainty from imperfect sensors, the *behavior planning* of the lane change assistant has to be predictive, consistent, deterministic, and punctual (cf. [UM13]). Therefore, the *behavior planning* has not only to consider the current and past system states but also predict future system states and estimate the benefit of possible actions for future states under real-time constraints. In [UM15b], Ulbrich et al. propose a planning framework which allows the behavior planning of lane changes in uncertain, mixed-integer state spaces by using partially observable Markov decision process (POMDP) (cf. [KLC98]). A reward model determines a reward for each possible action in a given system state while a prediction model predicts future system states based on the application of these actions on the given system state. As a result, an optimal tactical action for current and future system states is assessed and commanded to the trajectory planning module of the stabilization.
Markov decision processs (MDPs) are a general model for planning and decision making problems under uncertainty (cf. [Bel57]). The planning incorporates the modeling of rewards $r \in R$ for the application of actions $u \in U$ in a state $x \in X$ and a prediction model which predicts the state x_{i+1} for the application of action u in state x_i. The goal is to find an optimal sequence of actions $R_T = E\left[\sum_{\mathcal{T}=0}^{T} \gamma^{\mathcal{T}} * r_{\mathcal{T}}\right]$ which maximizes the expected reward $r_{\mathcal{T}}$ with a discount factor $\gamma^{\mathcal{T}}$ for the time horizon T. Therefore, MDPs can be applied to the planning of lane changes but extensions for predictability and real-time computation are necessary. Not all true states of the system are directly observable (cf. Section 3.3.1.2). POMDPs address this issue by introducing the belief $bel(x_t)$ about the state $x \in \mathcal{X}$ in which the system resides at time t. POMDPs model the sets of states X, actions U, and observations Z as value-discrete. For the lane change assistant, the state and observation have to be high-dimensional, mixed-integer spaces with uncertainties as they occur in the real world. Only the set of actions R remains value-discrete as discrete choices of actions. [UM13]

POMDPs have a high computational complexity which generally impedes their usage in real-time control applications [UM15b]. For the lane change assistant, the efficiency of action selection by POMDP can be improved by incorporating knowledge about the domain. Following [UM15b], this domain knowledge encapsulates:

1. A high planning accuracy is only evident for the immediate future. Long-time predictions or even infinite planning horizons are not necessary because the environment autonomously changes without any control of the automated ego vehicle. For the lane change assistant, the planning horizon for lane changes ranges between 100 ms and 30 s [UM15b].

2. The set of action alternatives for the automated ego vehicle is finite. There are many variations of the same maneuver but only a few mutually exclusive, discrete action alternatives exist. For the lane change assistant, the set of actions U is finite and contains 13 discrete action alternative (cf. [UM15b]). These action alternatives are *lane change (LC)*, *FinishLc*, *PrepareLc*, *IndicateLc* and *AbortLc* to the left and right lane as well as *NormalDriving*, *AbortLcIndication* and *AbortLcPreparation*. *Preparation* describes all activities before a lane change including, e.g., the proper positioning in the current lane and *Indicate* the activation of the direction indicators.

3. A subset of system states $X_c \subset X$ is free of uncertainty. These states significantly reduce the complexity of the model and even rule out some action alternatives.

4. The planning accuracy becomes less important for future states because only the immediate, next action will be commanded to the trajectory planning module. Detailed plans for the future are not necessary as they will be recalculated in the next time step.

Tree-based policy evaluation is used for the planning of lane changes under the consideration of the available domain knowledge. The tree-based policy evaluation understands the state space as a tree of beliefs $bel\,(x)$ over states $x \in X$ and actions $u \in U$. Such a tree of beliefs is presented in Fig. 3.12 (cf. [UM15b]). The figure abstracts that one action might result in multiple state beliefs. The maximal height of the belief tree corresponds to the finite planning horizon T. [UM15b]
The root of this tree is the belief about the current system state $bel\,(x_0)$ which has been provided by the situation assessment (cf. Section 3.3.1.2.
Each sequential layer represents the possible beliefs $bel\,(x_{t+\tau})$ about states $x_{t+\tau}$ for the next time stamp $t + \tau$ which result from the application of actions $u^i \in U$. The application of an action $u^i \in U$ for a state belief $bel\,(x_t)$ results in a reward $r\,(bel\,(x_t)\,, u^i)$. In contrast to the single value prediction in classical POMDP (cf. [KLC98]), the reward r for behavior planning aggregates the lane change possibility, benefit, and gap selection dimensions in one single vector. This vector representation allows the estimation of each dimension's impact on the total reward. Various aspects of the system's state belief $bel\,(x_t)$ are maintained by a decision hysteresis.[UM15b]
Actions resulting in unreasonable policies (sequence of actions), violating, e.g., requirements or national traffic codes, can be ruled out immediately. In Fig. 3.12, the actions

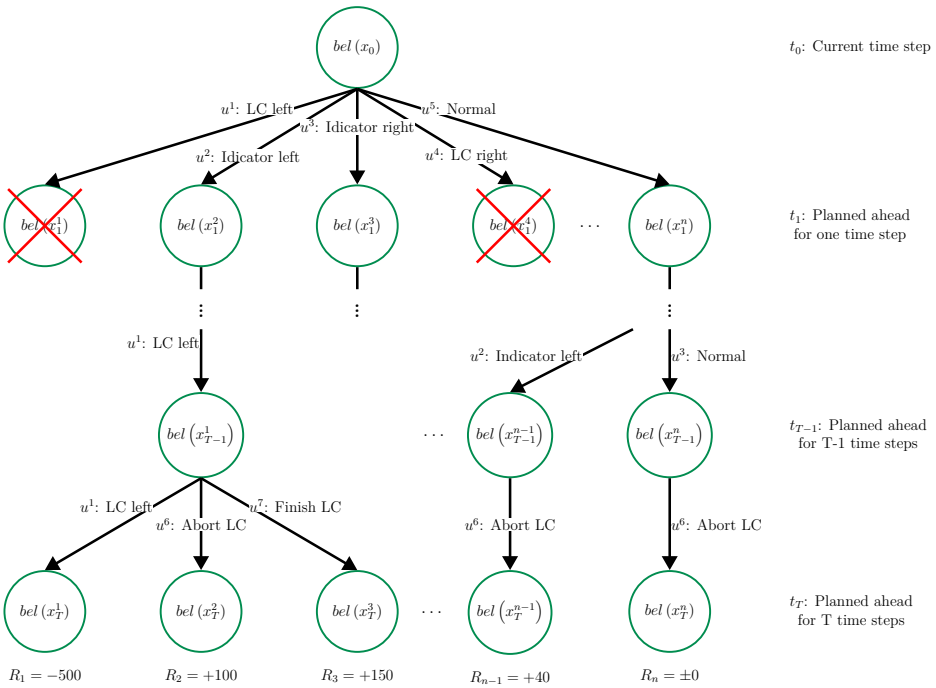


Figure 3.12.: Policy tree of (predicted) state beliefs and actions [UM13].

u^1 and u^4 for the current state belief $bel\,(x_0)$ are ruled out because lane changes must not be performed without its previous indications by flashing the direction indicators (action u^2 and u^3). This significantly reduces the complexity of the belief tree and its evaluation. [UM15b]

The prediction of future beliefs about system state can be depict as a function p : $bel\,(x_{t+\tau}) = p\,(bel\,(x_t)\,, u^i)$ based on the current state belief $bel\,(x_t)$ and an action u^i. The prediction has to incorporate various situations aspects, including, e.g., objects movements, vehicle interactions, and vehicle behavior. All these aspects have to represented by models that offer sufficient accuracy in reasonable time. For the lane change assistant, an improved version of the intelligent driver model of Shen et al. (cf. [SJ12]) has been used. The model estimates the movement and position changes of vehicles in the longitudinal direction by predicting their longitudinal acceleration and velocities. For lateral movement, the prediction model assumes vehicles to maintain their lateral offset towards their current lanes. [UM15b]

The finite planning horizon T for the lane changes assistant bounds the depth of the belief tree. For further reduction of the computational requirements, the size of time steps within the belief tree progressively increases with each step until the planning horizon T is reached. If $n = 1, \ldots, n$ is the number of time steps for the planning horizon T and δt the initial time step, then $\langle \tau_1 = 1 \cdot \Delta t, \tau_2 = 2 \cdot \Delta t, \ldots, \tau_n = n \cdot \Delta t \rangle$. This multi-resolution approach to the behavior planning allows to reduce the computational complexity further.[UM15b]

The selection of the action $u^i \in U$ for the current time step t is based on the past with the highest total reward throughout the belief tree. The total reward R_j is calculated for each

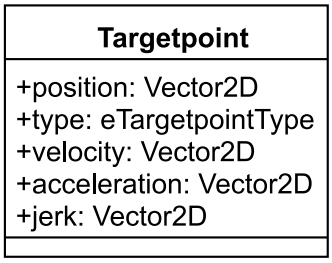
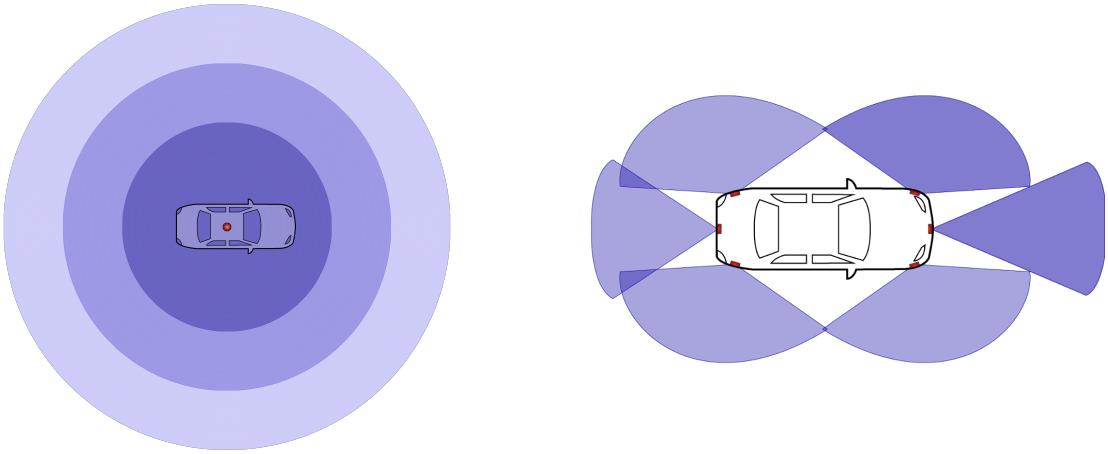


Figure 3.13.: UML class diagram of the target point.

path through the belief tree from the root to the fringe nodes $\langle (x_0), \ldots, bel(x_m) \rangle$ where m corresponds the planning horizon T (cf. Fig. 3.12). From all paths, the path with the highest total reward is selected and its action u^i for the time step $t = 0$ is transformed into a *target point* — a geo-spatial point in front of the vehicle (cf. Definition 3.7). A *target point* in the ego lane or no target point at all indicate that no lane change is envisaged by the behavior planning while a target point in one of the neighbor lanes indicate a lane change to that particular lane.

Definition 3.7 (Target Point)**.** *A target point is a geospatial point in front of the vehicle and represents the target destination for the trajectory planning by the stabilization.*

As shown in Fig. 3.13, geo-spatial position of a target point is enriched by further information about envisaged velocity, acceleration, and its type. The attributes *velocity* and *acceleration* enable ADAS functions to pass a target velocity and target acceleration to the trajectory planning of the stabilization (cf. Fig. 5.1). These parameters are essential for the ACC but are also used for lane changes by the lane change assistant.
The *type* of a target point defines how the ADAS function envisages the stabilization modules to process the target point. The enumeration *eTargetpointType* defines the possible types for target points and includes e.g., following behind other vehicles (*Follow*), drive at desired velocity (*Velocity*), and drive to a defined position (*Position*) on the ego lane. For lane changes to right resp. left neighbor lane, two identical sets of types are defined (*LaneChangeLeft* and *LaneChangeRight*). Each set encapsulates a *Follow*, *Velocity*, and *Position* type. While the *Velocity* type correlates to lane changes with defined velocities, the *Follow* and *Position* types relate to lane change with respect to a defined object resp. position on the particular neighbor lane.
As shown in Fig. 3.3, multiple ADASs are simultaneously executed in an automated vehicle. For the highway pilot, functions, e.g., ACC and LKAS are deployed alongside the lane change assistant in order to autonomously drive behind other vehicles in the current lane. Each function independently processes a specific action for the current situation and outputs a corresponding target point. From the set of target points one final target point is processed and commanded to the trajectory planning of the stabilization (cf.



(a) Single LIDAR sensor. (b) Multiple limited LIDAR sensors.

Figure 3.14.: Configuration of LIDAR Sensors for a 360° field of view.

selection in Fig. 3.3). The trajectory planning processes the trajectory for the automated ego vehicle from the final target point. The final trajectory might not incorporate a lane change to a neighbor lane even though the lane change assistant has planed a lane change. [Ulb+16]

3.3.2. Technical Architecture

Beside the functional architecture, a technical architecture is specified in the system design. The technical architecture defines a set of hardware components, e.g., sensors, actuators, and ECUs, which are required for the execution of the functional components. The technical architecture of the lane change assistant integrates a sensor configuration (cf. Section 3.3.2.1) with a execution platform for the system's functional components (cf. Section 3.3.2.2).

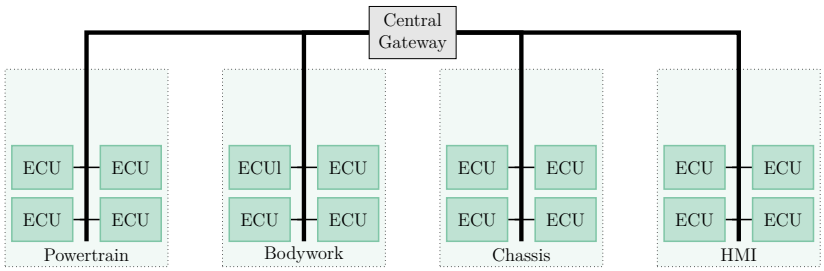
3.3.2.1. Sensor Configuration

The lane change assistant has to sufficiently perceive the vehicle's environment in order to make safe decisions about lane changes. While an ACC solely considers preceding vehicles in the same lane, the lane change assistant has to consider vehicles positioned all around the automated ego vehicle in near and far distances.
Following the functional architecture (cf. Fig. 3.3), one sensor might be sufficient to fully perceive the automated vehicle's environment. LIDAR sensors exist, e.g., Velodyne HDL-64E (cf. [MS11]), which are able to provide a 360° horizontal view around the automated vehicle (cf. Fig. 3.14a). In Fig. 3.14, each sensor unit is represented as a rectangle or circle for its position in the vehicle and its occupancy of the vehicle's environment — *range of view* — is depicted as a colored cone. Cones of multiple sensors may overlap representing the overlapping of their perception fields. The view around the vehicle is only limited in the vertical direction by the aperture of these sensors. For the

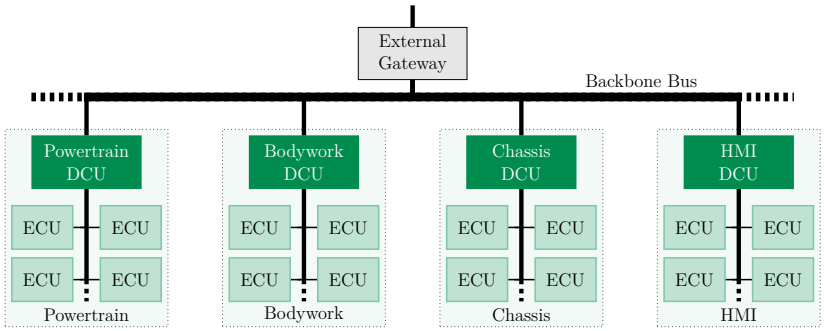
Velodyne HDL-64E sensor the vertical field of view ranges from $+2°$ to $-24.9°$. Such sensors can detect all vehicles in the vehicle's vicinity.
Another approach for perceiving the environment of the vehicle is the installation of multiple sensor modules around the vehicle with limited fields of view (cf. Fig. 3.14b). A single sensor module with limit fields of view is incapable of perceiving the complete vicinity of the automated ego vehicle. An aggregation of data from limited sensor modules all around the vehicle enables the generation of a complete view of the vehicle environment. The aggregation of data from multiple sensor modules requires to identify and relate the incidences of environmental objects in the fields of view of multiple sensors — especially for sensors with overlapping fields of view (cf. Fig. 3.14b). LIDAR sensors are not only able to detect large dynamic objects, e.g., other vehicles, but also smaller objects, e.g., road markings. Therefore, LIDAR sensors allow the detection of the road and its driving lanes (cf. [Zei13]) necessary for the planning of lane changes. Other prevalent sensors for the perception of the vehicle environment are RADAR, ultrasonics, and camera sensors.
The data generated by sensors for a $360°$ view around the automated ego vehicle is enormous. With the increasing level of automation (cf. Fig. 2.4), more diverse sets of sensors have to be deployed in order to perceive the complete vicinity of automated ego vehicles consistently. Additional ECUs have to be introduced in automated vehicles for processing requirements of the increasing sensor data (cf. environment perception in Fig. 3.3) and their complex analyses by automation functions. The following section describes the architecture of such an execution platform for automated vehicles.

3.3.2.2. Execution Platform

Prior to ADAS, E/E architectures of vehicles were subject to different vehicle subsystems (domains), e.g., powertrain, bodywork, chassis, or HMI (cf. [SZ13]). Each subsystem has been developed independently from the other subsystems. These architecture emerged with the introduction of stabilization functions, e.g., ABS and ESC. These E/E architectures distributed the vehicle functions over more than 80 ECUs in vehicles. ECUs are robust processing units for embedded systems which withstand large temperature differences but offer only limited processing and data storage capabilities. The vehicle functions are common their ECUs are highly optimized in order to save costs. For the exchange of data, ECUs of subsystems are connected with each other by communication bus systems (cf.[Bro03]), e.g., CAN, media oriented systems transport (MOST), or FlexRay (cf. [Fle05]). Gateways, which can be seen as specialized ECUs, interconnect the different subsystems and organize the message transfer between of their sub-networks (cf. [Kim+08]).
Figure 3.15a (cf. [SZ13]) shows the basic structure of such E/E architectures. Blocks denote hardware components, e.g., ECU or sensor units, while lines between blocks represent physical connections between hardware components. We do not restrict the data flow directions between hardware components over physical connections and do not differentiate the type of physical connections — if they are proprietary direct connections or standard bus systems, e.g., CAN or MOST. In case the type of physical connection is



(a) Execution platform aligned by vehicle subsystems.



(b) Execution platform with domain controller units.

Figure 3.15.: Evolution of E/E architectures.

defined and important, we label the corresponding connection with its type. In reality, E/E architectures vary between car manufacturers and are more complex then presented in Fig. 3.15a (cf. [Wed16]).

The lane change assistant is more processing-intensive than vehicle functions for the vehicle stabilization, e.g., ABS and ESC, and require information from all vehicle subsystems and even external sources, e.g., road infrastructure or Internet services (cf. [Ben+14; Ger+14]). Recent E/E architectures in the automotive domain introduce domain control units (DCUs) in subsystems for the execution of today's and future complex vehicle functions, e.g., the lane change assistant (LCA) (cf. Fig. 3.15b) (cf. [Som+13]) DCUs address the requirements for high computational power and large storage capabilities under reasonable costs by incorporating hardware components from customer electronics, e.g., ARM central processing units (CPUs) or Ethernet networks.

Each DCU is connected to the existing ECUs and sensors of its subsystem via the existing bus networks. The DCU acts as the master of its subsystem and gathers, e.g., sensor data, and forwards this data to other functions within its subsystem or other subsystems. The DCUs of different subsystems exchange their data via a high speed backbone, e.g., Ethernet (cf. Fig. 3.15b). The lane change assistant is executed on the DCU in the chassis resp. safety subsystem but has access to all required data for the processing of its environment representation and decision making (cf. Fig. 3.3). The result of its decision making is transmitted for execution to the other subsystems via the high-speed network.

The idea of centralization within automotive E/E architectures can be evolved to the integration of multiple subsystems on one single DCU or to an architecture with only one

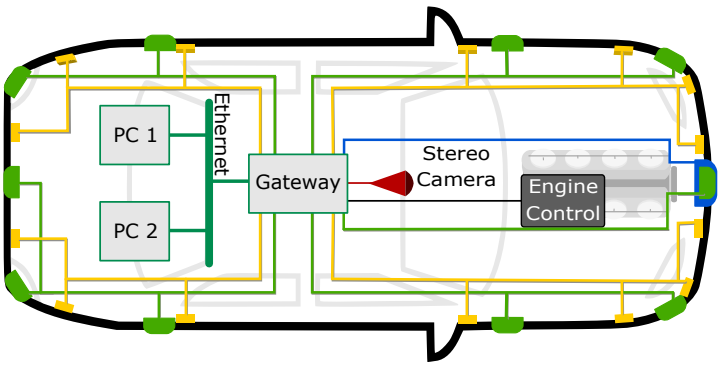


Figure 3.16.: Processing network of the prototype vehicle.

large vehicle control unit which integrates and processes all complex and cross-cutting functions of one vehicle. Furthermore, functions are envisaged to communicate and share data with Internet services outside the vehicles (cf. [Ger+14]).

None of the prototype vehicles (cf. Fig. 3.16) in the case study incorporate one of the presented E/E architectures (cf. Fig. 3.15). Instead, the original hardware equipment of a production vehicle is extended by two customer computers, a gateway computer, and an Ethernet bus. The original hardware components of the production vehicle, e.g., ECUs and CAN buses, process the original vehicle functions of the prototype vehicle, like ESC or automatic parking assistant. The two customer computers are introduced as the execution platform for the software implementation of the highway pilot and lane change assistant (cf. Fig. 3.16). The environment perception of the lane change assistant is executed on the first computer while the *situation assessment* and *behavior planning* of the lane change assistant and other ADAS, e.g., ACC and LKAS, are executed on the second computer (cf. PC 1 and PC 2 of Fig. 3.16).

The two computers are connected via an Ethernet network with each other for the exchange of function internal data, e.g., the *scene*, and with the gateway computer for the communication with vehicle sensors and actuators, e.g., ultrasonic sensor, steering, brakes, and engine (cf. Fig. 3.16). The gateway connects the execution platform of the lane change assistant with the existing communication systems of the vehicle, e.g., CAN and FlexRay. Sensors, which have been installed particularly for the lane change assistant, are also connected to the gateway computer. Sensor data is received from the vehicle sensor and converted by the gateway into the appropriate data format for the processing of *environment perception*. Actions by the lane change assistant are transferred via the gateway to the vehicle actuators as their inputs.

The execution platform of the prototype vehicle for the lane change assistant does not focus on sustainability but adaptability and changeability in order to enable efficient evaluation, adaption, and improvement of the system and its algorithms early in the development. The customer computers allow fast development iterations of the lane change assistant without flashing its software code for every new version. Flashing ECUs is a time consuming and costly task — especially under consideration of the typical high frequency of changes to functional components in the early stages of system development.

As the necessity for changes of the lane change assistant decreases in later stages of the development, the hardware platform of the new prototype vehicles may converge to the final hardware used in production vehicles.
The interaction of the lane change assistant with its environment imposes potential risks for itself, its passenger, and all objects and persons in its vicinity. Therefore, it is necessary to address the correct implementation of safe system behavior in order to mitigate these risks. The presented lane change assistant has been straightforwardly designed. The presented sensor configuration and execution platform are not applicable for automated driving under consideration of safety aspects (cf Definition 2.2). Especially for higher level of automation (cf. Fig. 2.4), a single sensor configuration (cf. Fig. 3.14a) would represents a *single point of failure* for a complete loss of the environment perception. Potential failures and risks have to be identified and mitigated for the lane change assistant in order to ensure the *safety* of the lane change assistant during operation in the real world (cf Definition 2.2). The safety analysis is exemplarily performed in the following section on the presented system design concept for the lane change assistant.

3.4. Safety Analysis

The lane change assistant is a safety-critical system which makes *mission-* and *safety-critical* decisions on behalf of humans while driving through traffic. These decisions may potentially threaten the safety of other vehicles, objects, and persons by contributing to accidents [Bei12]. For the operation in public traffic, all decisions of the automated vehicle must be safe in all common and critical situations as well as in the presence of internal system faults and loss of hardware components. *System faults* must not lead to critical situations in which any person or object is harmed (cf. Definition 2.12).
An established safety principle for safety-critical systems, like the lane change assistant, is the assumption that these systems are unsafe unless convincingly argued otherwise [KW16]. The safety of these systems has to be shown and not to be assumed. Therefore, product liability, as well as the functional safety, has vital importance for car manufacturers. National authorities — at least in Germany — force car manufacturers to sufficiently prove and document the safety of their vehicles and systems before their commercialization.
The safety analysis uses established methods, e.g., hazard and operability study (HAZOP), fault tree analysis (FTA), and failure mode and effect analysis (FMEA) among others (cf. [Ise11]), in order to systematically identify, analyze, and classify hazards for these investigated systems. A *hazard* is a system state or event which impose potential harm for objects and persons (cf. [Int09a]) and generally corresponds to a malfunction of the system (cf. Definition 2.141). For example, the malfunction of vehicle brakes imposes potential harm for the vehicle's passengers as well as other vehicles, their passengers, and pedestrians in the vicinity of the malfunctioning vehicle due to the increased risks of collisions. For each hazard, a least one corresponding safety measure has to be defined which reduce the increased risks by the hazardous system to an acceptable level in order to maintain the system safety (cf. [SZ13]). Otherwise, the lane change assistant is unable to meet the legislative standards concerning, e.g., producer and product liability.

Definition 3.8 (Hazard)**.** *A Hazard is a system state or event which impose potential harm for objects and persons (cf. [Int09a])*

In the automotive domain, the safety standard ISO 26262 is the predominant safety standard for the engineering of safety-critical systems — include their safety analysis (cf. [Int09c]). National authorities widely accept results from a safety analysis following the safety standard ISO 26262 as evidence for the overall safety of vehicles and their systems. The safety standard ISO 26262 standardizes a safety lifecycle which addresses the systems' *functional safety* by systematically construct arguments for the absence of unreasonable risk due to hazards caused by malfunctioning behavior of E/E systems for automobiles up to 3.5 t. Such an argumentation is denoted as *safety case* by safety standard ISO 26262 (cf. Definition 3.9).

Definition 3.9 (Safety Case)**.** *"An argument that the safety goals for an item [, the system (part) under investigation,] are complete and satisfied by evidence compiled from work products of the safety activities during development" [Int09a].*

Any safety analysis commences with the definition of the system under investigation (named *item* in the safety standard ISO 26262) including the definition of system parts, environment situations, and events which are explicitly excluded from the analysis. Here, the system under investigation is the automated vehicle with the focus on the lane change assistant as the subsystem. The lane change assistant as the item under investigation is analyzed in the *hazard identification and risk assessment* for the identification of safety-critical hazards and the definition of corresponding safety measures (cf. [Bir+13]). In the following sections, the *hazard analysis and risk assessment* exemplary performed. An exemplary FTA is created for the lane change assistant on the example of a collision with a vehicle in front while changing the lane.

3.4.1. Hazard Analysis and Risk Assessment

The hazard identification by the *hazard identification and risk assessment* requires experts to define potential operational situations and operating modes for the lane change assistant. Operation situations and operation modes are aggregate in hazardous scenarios (cf. Definition 3.10). The behavior of the lane change assistant is analyzed for the hazardous scenarios in order to identify hazardous events. A *Hazardous event* is a combination of a system hazard and operational situation [Int09a].

Definition 3.10 (Hazardous Event)**.** *A* Hazardous event *is a combination of a system hazard and operational situation [Int09a].*

Each identified hazardous event is evaluated and classified based on its severity — ranging from no injuries to life-threatening injuries —, probability — ranging from incredible to highly probable —, and controllability — ranging from controllable in general to

uncontrollable. The classifications for severity, probability, and controllability result in the joint classification as automotive safety integrity level (ASIL) for the hazardous event (cf. [Int09a; Int09c]). In conjunction with the ASIL classification, a high-level *safety goal* is defined for each hazardous event from the *hazard analysis and risk assessment*. A *safety goal* represents a top-level safety requirement which defines functional objectives required to mitigate the corresponding hazard and its risks (cf. Definition 3.11). However, the technical implementation of necessary safety measures is not addressed by safety goals (cf. [Int09c]). All safety goals are documented in the system's *safety concept*. One safety goal may address more than one hazardous event but can only have one ASIL assigned. Therefore, the most severe ASIL is selected from the set of possible ASIL.

Definition 3.11 (Safety Goal)**.** *A safety goal represents a top-level safety requirement which defines functional objectives required to mitigate the corresponding hazards and risks but does not address the technological implementation of necessary safety measure (cf. [Int09a]).*

For the lane change assistant, hazardous scenarios include collisions with other traffic participants or infrastructure objects resulting in casualties or even fatalities. While changing lanes on highways, the automated vehicle has to consider other traffic participants, e.g., vehicles driving in front of the automated ego vehicle, next to it, or approaching from behind, and objects and must not collide with them. Each of these dangerous scenarios can be investigated independently but lead to similar high-level safety goals — *exclude any collision with other traffic participants*. In all these scenarios, a collision imposes the risks for potential fatalities. The ASIL classification for these hazardous events are predominantly ASIL D (cf. [Int09c]). This classification especially applies to higher levels of automation in which the driver is not available to take over the control of the vehicle and mitigate the hazardous event (cf. Fig. 2.4).
Safety goals define top-level functional objectives for the mitigation of hazards but do not define necessary safety measures in terms of technological solutions. The safety standard ISO 26262 introduces functional and technical safety requirements for the refinement of safety-goals and allocation of necessary safety measures to (sub) components of the system. Functional and technical safety requirements are described in the following section.

3.4.2. Functional and Technical Safety Requirements

The safety standard ISO 26262 introduces *functional* and *technical safety requirements* for the refinement of safety-goals and allocation of necessary safety measures to (sub) components of the system. *Functional safety requirements* specify necessary safety measures independent of their technical implementation (cf Definition 3.1). Each functional safety requirements is allocated to one or more functional components of the system's architecture. The set of allocated functional safety requirements for one functional component defines the *safety measures* which this component has to implement. All functional safety requirements are documented in the *functional safety concept* (cf. [Int09c]).

Definition 3.12 (Functional Safety Requirement)**.** *"The specification of implementation-independent safety behavior, or implementation-independent safety measure, including its safety-related attributes" [Int09a].*

In the development, functional components are designed and implemented as software and hardware components (cf. Section 3.5). Therefore, functional safety requirements are further transformed into technical safety requirements which are allocated to the software and hardware components of the system. Technical safety requirements refine the functional safety requirements by describing a concrete technical implementation of the corresponding safety measures under the usage of technological solutions, e.g., diagnosis functions or hardware redundancy. All technical safety requirements are first documented in the *technical safety concept* and later by the corresponding *software* and *hardware safety requirements specifications*. These specifications contain the information about the verification and validation of safety requirements and the corresponding implementation (cf. [Int09e] and [Int09f]). Safety requirements resemble additional requirements for the system that have to be considered in the design, implementation, verification, and validation of the lane change assistant (cf. [HRS98]).

Definition 3.13 (Technical Safety Requirement)**.** *"The requirements derived from the associated functional safety requirements to provide their technical implementations" [Int09a].*

A suitable approach for the refinement of safety goals to (sub) components among others is the FTAs. The FTA enables to refine the *safety goal* — not collide with other vehicles — to functional and technical safety requirements and allocate them to components of the lane change assistant. In the following section, a FTA is presented for the hazardous event of a *collision with a preceding vehicle while changing the lane*.

3.4.3. Fault Tree Analysis

FTA is a deductive failure analysis method which uses binary logic to analyze how low-level fault events of the system components contribute to hazardous events of the overall system (cf. [Lee+85]). Subordinate Faults are related via *AND* or *OR* gates with their superordinate fault. Subordinate faults lead to the superordinate fault independently (*OR*) or have to occur concurrently (*AND*). Therefore, the FTA considers *single point of failure* as well as combinations of faults — *multiple points of failure* — that may result in hazards.
For the lane change assistant, a scenario is exemplarily analyzed. The scenario examines the collision of the automated vehicle with a preceding vehicle driving in front while it is changing the lane. Figure 3.17 presents the example FTA for this scenario. The top-level event is the collision with the vehicle driving in front which corresponds to the negation of the safety goal; *the automated vehicle must not collide with other vehicles while changing the lane*.

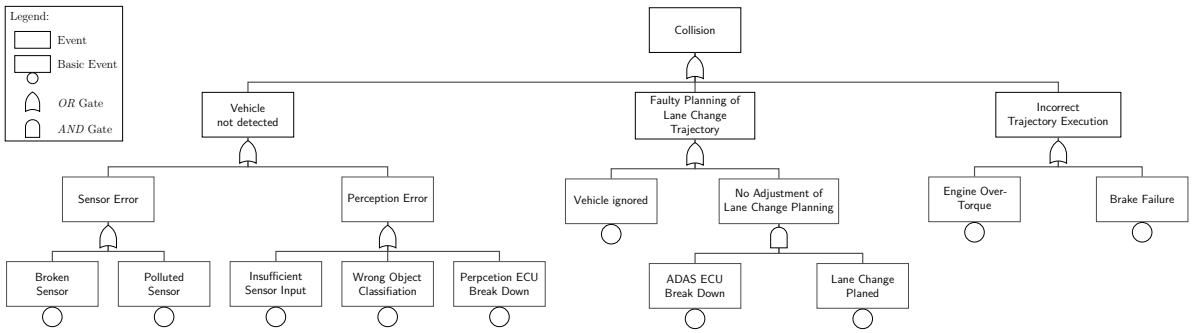


Figure 3.17.: Exemplary fault tree for collision with a vehicle in-front.

As shown in Fig. 3.17, the FTA for the lane change assistant commence with a top-level fault event — the negation of the safety goal — and decompose this top-level event into tree of minor fault events (cf. [Eri05]). The leaves of the faults tree represent basic, external, and undeveloped fault events. Basic fault events are allocated to components of the system and, therefore, represent faults of these components (cf. Definition 2.12). The identification of component faults enables the definition of appropriate *safety measures* for these components. Safety measures may include methods, e.g., fault detection and failure mitigation, fault tolerance methods, as well as the transition to *safe states* among other solutions (cf. [Ise11]). A *safe state* is a state of the system in which any risks for the system and its environment is excluded (cf. Definition 2.8).
The presented FTA in Fig. 3.17 is an example and is not supposed to complete. Other faults may be relevant for the lane change assistant. Nevertheless, the minor faults of the FTA for the collision with a preceding vehicle while performing a lane change are described in the following sections.

3.4.3.1. Faults for the Perception of the Vehicle

As depict by the most left subtree of the example FTA in Fig. 3.17, faults prohibiting the detection of objects and vehicles could be an error of the sensor hardware or of the perception functions (cf. Fig. 3.3). A sensor may break down and provide no data at all, or a sensor may be polluted and only provide partially valid sensor data. The system would not be able to sufficiently monitor its vicinity considering the system architecture of the lane change assistant (cf. Fig. 3.3). Besides faults of the sensor hardware, perception functions may faulty processes correct data from the sensor. The provided data by a polluted sensor might not provide sufficient correct representation for the real world for the correct identification of the preceding vehicle by the perception functions. Another fault for the perception could be the wrong classification of the preceding vehicle. This could lead the *function specific scene augmentation* to ignoring the vehicle in front in *situation* (cf. Fig. 3.3). The third fault for the perception is the break down of the corresponding ECU (cf. Fig. 3.17). The loss of the ECU would prohibit any processing of sensor data. All described faults result in no or an incorrect representation of the current real environment for the *situation assessment* of the lane change assistant (cf. Fig. 3.3).

3.4.3.2. Faults for the Planning of Lane Changes

Besides the *environment perception*, also the *situation assessment* and *behavior planning* can introduce faults (cf. mid subtree of Fig. 3.17). One fault could be that a vehicle in front is omitted from the situation and the planning falsely assumes the space this vehicle occupies to be free. This fault corresponds to the fault of the perception concerning the misclassification of the preceding vehicle. Another fault for the lane change planning is the omitted any adaption of currently executed lane changes even though newer information about the changed environment situation is available. In the normal case the lane change assistant will adjust its planned lane change to the new environmental information but if the ADAS ECU is broken down the trajectory executed by the stabilization cannot be adjusted. In contrast to all other faults in Fig. 3.17, the subordinate faults for *no adjustment of lane change planning* are joined by an *AND* gate. The break down of the ECU has to occur when a lane change is already planned. For all other cases, *OR* gates are used as these faults lead to the top-level fault event — the collision — independently from other faults.

3.4.3.3. Faults for the Execution of Lane Changes

Even though the vicinity of the automated vehicle has been correctly perceived and the lane change assistant has correctly planed the lane change, collisions may still occur due to faults of actuators. As described by the right subtree in Fig. 3.17, the engine could get stuck and generate excessive torque or brakes fail and not produce the necessary brake force. Even if the lane change assistant has planed a collision-free trajectory, both faults will result in the automated vehicle driving too fast resp. not sufficiently decelerating in time in order to avoid the collision with the preceding vehicle.
All identified faults in Fig. 3.17 can be allocated to one of the high level components of the lane change assistant — *environment perception*, *situation assessment*, *behavior planning*, or *stabilization* (cf. Fig. 3.3). Based on the FTA, safety measures are defined that mitigate one or more faults and reduce the probability of the hazardous event — the collision — in order to increase the overall safety of the system. We omit the explicit definition of functional and technical safety requirements and their respective concepts, but we describe the impact of this safety analysis on the requirements analysis, system design, implementation, verification, and validation for the lane change assistant in the following section.

3.4.4. Result and Impact of the Safety Analysis

Under consideration of potential system and component faults (cf. FTA for the lane change assistant in Fig. 3.17), the processing chain for the decision making of the lane change assistant is not able to provide the sufficient safety guarantees that are required for such autonomous systems. For example, a sudden break down of a ECU or LIDAR sensor could lead to the complete loss of the lane change assistant's functionality while performing a lane change. This loss would impose potential harm for all objects and

persons in the vicinity of the automated vehicle. Until now, faulty system components had to be transferred to a *safe state* (cf. Definition 2.8), which prevents these components from further interaction with the remaining system. As the driver has been responsible for the vehicle control at all time, car manufacturers could still rely on the driver as the final safety instance to mitigate any hazardous situations. This safety approach is called *fail-safe*.
With higher levels of automation (cf. Fig. 2.4), the driver is not required to control and supervise the vehicle permanently. Vehicle systems have to ensure their safety and remain operational — even in the presence of system and component faults. At least one alternative processing path has to be introduced for these systems, which can take over the control of the vehicle in case the original function fails. For example, this can be a second version of the original function on its hardware platform or a safety function that provides sufficient but degenerated functionality, e.g., changing to the emergency lane. Safety functions with degenerate functionality must safely transfer the system into a *safe state* where any harm for persons and objects are excluded. This level of required system safety is called *fail-operational*.
In the following sections, the impacts of the safety analysis on the development activities for the lane change assistant are described.

3.4.4.1. Impact on the System Requirements

The safety analysis results in definition of the *functional* and *technical safety concept* with the identified *functional* and *technical safety requirements* among other results, e.g., verification, validation, integration, or testing plan. All requirements from the safety analysis can be considered as additional safety requirements which have to be added to the set of initial requirements from the requirements analysis (cf. Section 3.2). For the lane change assistant, safety invariants are defined along the functional and technical safety requirements. The safety invariants correspond to the high-level safety goals for the lane change assistant and represent requirements which the lane change assistant has to satisfy at all times. The following sections describe the additional safety requirements for the lane change assistant in more detail.

3.4.4.1.1. Safety Invariants

The safety goals identified in the *hazard analysis and risk assessment* (cf. Section 3.4.1) relate to system invariants which have to be satisfied by the lane change assistant at all times. In some cases, safety goals can be represented by a smaller number of high-level safety related invariants. Two main invariants can be defined for the lane change assistant:

- The lane change must not actively collide with other traffic participants while changing to another lane.
- If no lane change is performed, the driver or other assistance systems, e.g., ACC, will be responsible for the overall safety of the vehicle.

Table 3.10.: Requirements for the environment perception.

ID	Description
FR_11	The system has to monitor the quality and availability of all its input data.
FR_11_1	The system has to monitor the observability of the 360° field of view around the vehicle by sensor systems at all times.
FR_11_2	The system shall monitor and detect faults of not redundant sensors at all times.
FR_11_3	The system shall be able to detect delayed, distorted or faulty sensor data at all times.

While the first invariant addresses the safety of the lane change assistant and its decisions, the second invariant defines the scope of the safety assessment for the lane change assistant and redirects the safety responsibility to other systems or the driver. These invariants for the lane change assistant are exemplarily defined for the lane change assistant and are not supposed to be complete. In reality, a more diverse set of safety goals resp. invariants would result from the *hazard analysis and risk assessment* (cf. Section 3.4.1).

The safety goals resp. invariants address the external perceptible functionality of the lane change assistant but are not applicable to the definition of the internal structure of the system. For the system design and implementation, *functional* and *technical safety requirements* have to be defined. These safety requirements are added to the initial set of requirements from the requirements analysis.

3.4.4.1.2. Robustness of the Environment Perception

The lane change assistant makes safety-critical decisions on behalf of human drivers. These decisions may potentially harm other objects and persons. Therefore, the lane change assistant has to monitor itself and detect faults by its components in order to autonomously adapt in order to continuously ensure the safety for itself, objects, and persons in its vicinity. All decisions about lane changes by the lane change assistant rely on the sufficient perception of the vehicle's vicinity.

As shown in Table 3.10, the lane change assistant has to monitor its environment perception (cf. Fig. 3.3) in order to detect if sensors or perception modules are unable to provide an accurate 360° representation of the vehicle's vicinity (FR_11_1). Furthermore, the lane change assistant has to monitor the quality and availability of data for all sensors (FR_11) in order to detect delayed, distorted, faulty, or unavailable sensor data (FR_11_2). Component redundancy is particularly important for the sensors which perceive parts of the vehicle's vicinity that are not perceived by any other sensor. In case such sensor fails, the lane changes assistant is unable to detect any objects within this part of vehicle's vicinity (FR_11_3).

Table 3.11.: Requirements concerning the ability restrictions.

ID	Description
FR_12	The system has to monitor its functional skills and abilities at all times with an error rate of less than x errors per hour of the driving time.
FR_12_1	The system shall be able to detect whether a lane change is possible when there is no speed limit, and there is no object on the neighbor lane.
FR_12_2	The system shall monitor its object perception viewing range in its ego and immediate neighbor lanes.
FR_12_3	The system shall monitor its lane perception viewing range in its ego and immediate neighbor lanes.
FR_12_4	The system shall known the adjusting ranges of actuators for planning its maneuvers.

3.4.4.1.3. Technical Abilities and Restrictions

The lane change assistant has not only to consider the quality and availability of data provided by the vehicle's sensors but incorporates its technical limitations and restriction, e.g., sensor ranges and actuator adjusting ranges, in its decision making (cf. FR_12 in Table 3.11).
The lane change assistant has to consider the range of its sensors in all decisions about lane changes. Sensor ranges, which are insufficient for expected velocity differences between the automated ego vehicle and other vehicles on the road, impose the possibility of collision while changing lanes (FR_12_1). Very fast or very slow vehicles outside the sensor ranges might be perceived too late by the automated vehicle resulting in insufficient time to avoid collisions by sufficient deceleration.
Sensor ranges can be reduced by near objects that block the detection of lane sections and corresponding objects behind the objects (FR_12_2 and FR_12_3). These hidden lanes section and objects would be relevant for lane change decisions if the lane change assistant knows about these lane sections and objects. For example, a vehicle on the same lane behind the automated vehicle might block a fast approaching vehicle on the left lane. The execution of a lane change by the automated vehicle to the left lane in this situation would result in a collision with the approaching vehicle.
The executions of lane changes impose further restrictions for the lane change assistant. Maneuvers, e.g., changing the lane or braking, must not require actions of actuators that are beyond their adjusting range or impose excessive g-forces for passengers (FR_12_4). The weather, e.g., snow, can further influence and limit the safe adjusting ranges of actuators.
The functional and technical safety requirements extend the set of requirements from the requirements analysis (cf. Section 3.2) and have to be considered in all development activities for the lane change assistant (cf. Section 3.1.2). The design of the lane

change assistant (cf. Section 3.3) has to be extended in order to incorporate these safety requirements. The following section presents the changes for the functional and technical architectures.

3.4.4.2. Impact on the System Design

The additional safety requirements identified in the safety analyses have to be considered in the system design of the lane change assistant. Functional safety requirements impact the functional architecture of the lane change assistant while technical requirements impact the technical design of the system in terms of hardware. Therefore, the functional and technical architectures of the lane change assistant have to be adapted and extended to meet these additional safety requirements. As described in the following, this may incorporate additional ECUs, sensors, and software functions.

3.4.4.2.1. Impact on the Functional Architecture

For the safety of the lane change assistant, its intended functionality and its functionality in the presences of component faults and system failure has to be considered. The disabling of faulty components is not possible in all situations. In some situations, e.g., on the leftmost lane of a highway, a disabling of the powertrain impose the risks for collision with following vehicles. A second independent processing path has to be introduced for the lane change assistant in order to remain operational in case of system faults and in order to transfer the automated vehicle into a *safe state* (cf. Definition 2.8). One solution is the duplication of the lane change assistant. The second instance of the system would be available to take over the control of the vehicle if the primary instance fails. The functionality of the lane change assistant remains available in its full extent until another system or component fault occur for the second instance of the lane change assistant. Another solution is to introduce dedicated safety functions which offer a degraded functionality of the lane change assistant. The safety function will transfer the vehicle into a safe state if the lane change assistant fails. An example of such a safety function with degraded functionality would be the changing to the emergency lane of a highway.
One of the additional safety requirements requires a permanent and accurate 360° field of view around the automated vehicle (cf. FR_11_1 in Table 3.10). The initial sensor configuration for the lane change assistant (cf. Fig. 3.14) is not able to meet this requirement. However, this requirement is essential for the safety of the lane change assistant. In case a non-redundant sensor breaks down the complete or a part of the vehicle vicinity — depending on the sensor configuration — cannot be accurately perceived. Therefore the sensor setup has to be extended by additional sensors in order to achieve complete coverage of the vehicle's environment by redundant sensors. Even if one sensor fails, another sensor will remain available to perceive the corresponding part of the vehicle's vicinity.
The introduction of additional sensors requires an extension of the functional architecture. The data from each sensor has to be individually preprocessed because each sensor

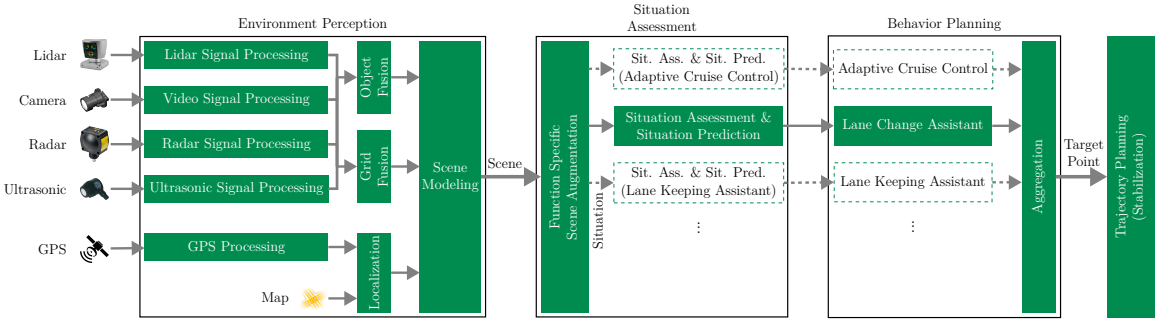


Figure 3.18.: Extension of the functional architecture following the safety analysis.

types requires different algorithms for its preprocessing. As shown in Fig. 3.18, a setup with multiple sensors of different types extends the environment perception by one preprocessing path per sensor types.
In the extended functional architecture (cf. Fig. 3.18), special processing steps are introduced for the detection of objects and the grid of free space around the vehicle (cf. [KH10]) before the information are incorporated in the *scene*. For the fusion of objects and grid, the position of each sensor in the vehicle has to be known in order to determine the positions of detected objects and free space correctly. The position of objects and the free space are determined in relation to the position of the automated vehicle in the map based on the vehicle's current GPS position.
With the *scene* as single interface between the *environment perception* and the *situation assessment* (cf. Fig. 3.18), the remaining parts of the lane change assistant — *situation assessment*, *behavior planning*, and *stabilization* — are not affected by the changes to a multi-sensor platform. Today's common architectures for autonomous vehicle systems towards automated driving are the result of year-long evolution, and the presented improvement of the architecture for the lane change assistant can be seen as one step in this evolution.

3.4.4.2.2. Impact on the Technical Architecture

As mentioned for the functional architecture, the safety requirements require a permanent and accurate 360° field of view around the automated vehicle (cf. FR_11_1 in Table 3.10). The resulting multi-sensor platform has to ensure that at least two sensors perceive every point in the vehicle's vicinity with overlapping fields of view in order to ensure that the vehicle's vicinity is accurately perceived even in case one sensor fails.
The multi-sensor platform of the prototype vehicles which recorded the evaluation data for the case study (cf. Chapter 8), incorporate LIDAR, RADAR, ultrasonic sensors, and stereo cameras. Figure 3.19 shows the configuration of this sensor set and its coverage of the vehicle's vicinity. The RADAR sensors are placed around the complete vehicle. Together with ultrasonic sensors, the RADAR sensors guarantee the exhaustive environment perception all around the vehicle. While the RADAR sensors are specifically installed for the lane change assistant, the ultrasonic sensors have been part of the original production vehicle as parking sensors. A large number of diverse sensors — LIDAR

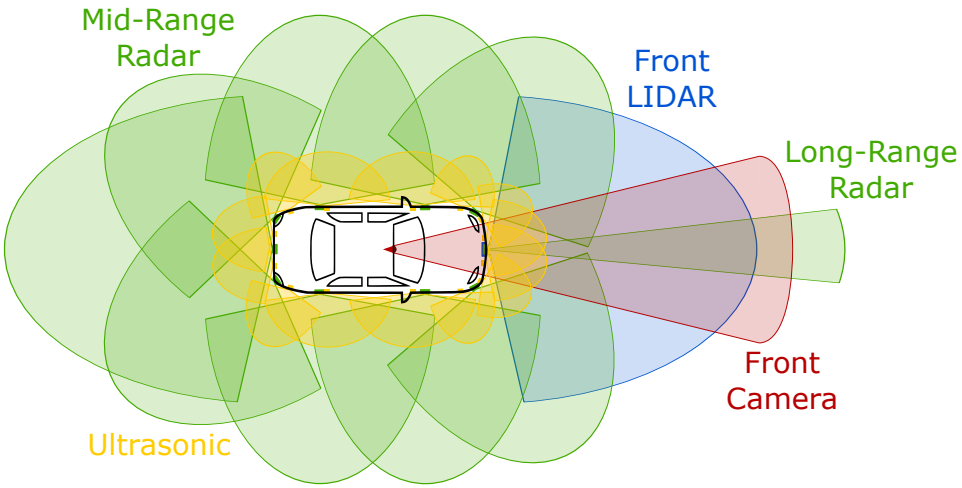


Figure 3.19.: Multi-sensor platform of the prototype vehicle.

sensors, long-range RADAR sensors, stereo camera, and ultrasonic sensors — is used for the perception of the road ahead. This configuration results in a highly redundant perception of the vehicle's environment in front by a diverse set of sensor types. Even if one sensor fails, the sensor setup of the prototype vehicle will ensure the accurate 360° view around the automated vehicle due to the overlapping occupation of the vehicle's vicinity by different sensors. Two or more sensor faults are required in this configuration for a partially insufficient perception of the vehicle environment.
Multi-sensor platforms enhance the general robustness of the environment perception. Each sensor type measures the world by a different type of physical magnitude. Stereo cameras passively use the ambient lighting to create an image of the environment while RADAR and LIDAR sensors actively use radio resp. laser waves to detect objects in the vehicle's vicinity. All sensor systems are subject to deviations and uncertainty (cf. [Ulb+15]), but each sensor type has specific deficiencies perceiving objects in certain environmental situations due to their psychical principles. Cameras might not be able to perceive and distinct objects correctly which have the same coloring or shading. In some case, dynamic objects may blend in with their environment[2]. RADAR sensors sometimes have difficulties to detect pedestrians correctly (cf. [Pat+17]). The multi-sensor platform enables to compare data from sensors of different types about the same area of the vehicle's environment with each other and to detect and purge perception errors of each sensor type. This comparison results in an environment representation with enhanced accuracy.
Additional to the extension of the sensor configuration, the execution platform of ECUs for the processing of functional (software) components has to be adapted.
Safety functions must not be executed on the same hardware components as the supervised functional components. Otherwise, a fault of relevant hardware components may compromise the original functions of the lane change assistant, and its safety fallback.

[2]https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk (Accessed: 12/06/2018)

The safety fallback — the second instances — would be incapable of providing their necessary safety functionality in order to retain the vehicle's safety. Therefore, the functional components of the primary lane change assistant (cf. Fig. 3.18) and its safety fallbacks have to distributed over the existing hardware platform In case a sufficient separation of system functions and safety functions cannot be achieved on the existing hardware platform of lane change assistant, additional hardware components, e.g., ECUs and communication buses, have to be introduced to the hardware platform. A direct approach to ensure the separation between nominal functionality and safety function is the introduction of dedicated hardware components for the safety functions, like the duplicated lane change assistant or the safety functions.
The next activity for the development of the lane change assistant after the safety requirements have been considered in the overhauled system design is the system implementation. The system implementation has to consider the extensions to the functional and technical system design for the safety requirements from the safety analysis. The impact of the safety analysis on the system implementation is described in the following.

3.4.4.3. Impact on the System Implementation

Following the extended system design, components of the functional architectures are implemented mainly as software components while components of the technical architecture are developed as hardware, e.g., sensors or ECUs. In either case, the measures defined by the safety analysis have to be considered in order to detect and mitigate systematic failures in the software implementation as well as random faults within the hardware components. A detailed description of the implementation is given in Section 3.5.

3.4.4.3.1. Impact on the Software Implementation

Each component of the functional architecture (cf. Fig. 3.18) is further refined and implemented as software components. Technical safety requirements assign to these functional components have to equally refined as software safety requirements to these software components. Besides the nominal functionality, each software component has to implement the safety measures defined by its assigned safety requirements.
The ISO 26262 recommends for each ASIL a set of recommended development methods, practices, and principles for the system design and its implementation. These methods have to be considered in the development of the lane change assistant and include, e.g., formal notations for system architecture and components, programming guidelines, like non-dynamic objects or limited use of pointers, and the application of documentation, modeling, and software tools (cf. [Int09f]).

3.4.4.3.2. Impact on the Hardware Implementation

Similar to the implementation of the functional architecture, the implementation of the technical architecture has to incorporate the safety measure defined by the assigned technical safety requirements. The safety standard ISO 26262 describes possible fault detection mechanisms, e.g., self-testing, checksums and signatures, and code protection

(cf. [Int09e]). In case the original hardware architecture is incapable of satisfying the technical safety requirements, the hardware architecture has to be extended by an additional hardware component, e.g., additional sensors or ECUs.
Its verification and validation follow the implementation of the lane change assistant. In the verification and validation (V&V), the correctness of the software and hardware is verified considering the extended set of requirements — including the safety requirements from the safety analysis. The impact of the safety analysis on the verification and validation is discussed in the following section.

3.4.4.4. Impact on the Verification and Validation

In the safety analysis, plans for the verification, validation and the assessment of the lane change assistant and its safety are defined in the *safety analysis* in addition to to the safety goals, requirements, and safety measures for functional and technical components (cf. [Int09d]). The plans define strategies for the verification, validation, and assessment of all development artifacts of the lane change assistant to evaluate their correctness and safety. These plans define all V&V activities throughout the development and their specific requirements, e.g., the documentation of V&V results. In general, results from these verification, validation, and assessment activities are documented in reports. These reports are the foundation for the argumentation in the certification process of the lane change assistant. The following section examines the impact of the safety analysis on the verification, validation, and safety assessment in more detail.

3.4.4.4.1. Verification

The verification for the lane change assistant defines the time of verification activities in the development process and their scope. The safety standard ISO 26262 defines various verification methods in dependency the ASIL of development artifacts. Examples of possible verification methods are designed verification, safety analyses as well as hardware, software, and item integration testing. Each of these activities addresses a different development artifact. While the design verification evaluates the functional and technical architectures from the system design, software, hardware, and integration test address the different implementation artifacts of the lane change assistant. Nevertheless, all activities will evaluate if the corresponding development artifacts meet their respective specifications and requirements. For the lane change assistant, we only considered the verification of the system design by reviews and the verification of implementation artifacts by reviews and tests.
The correct implementation of the lane changes assistant requires the system design to sufficiently consider all necessary functionalities which are required for the safe operation of the lane change assistant in the real world. Reviews verify the functional and technical architectures. In reviews, multiple engineers assess and evaluate the architectures in order to detect missing functionalities or insufficiently considered requirements. A particular focus is on the safety requirements obtained from the safety analysis (cf. Section 3.4).

The safety requirements from the safety analysis not only have to be considered in the functional and technical architectures but have to be correctly implemented. In implementation reviews, at least two engineers informally evaluate the design and implementation of software and hardware components for the lane change assistant. These reviews will determine if all components from the functional and technical architecture and their corresponding requirements have sufficiently considered in the implementation.

The software components of the lane change assistant are primarily verified in software tests — unit tests, integration tests, and system tests (cf. [Int09f]). For each software artifact, a set of test cases is defined. These test cases verify the functionality of each artifacts in regard to its requirements (cf. Section 2.2.2.7). The test cases have to sufficiently consider the additional safety requirements and safety measures from the safety analysis. In case the initial set of test cases does not sufficiently address the additional safety requirements for the corresponding artifact, new test cases have to be defined. Fault injecting methods (cf. [Ise11]) can be incorporated in tests to trigger relevant components faults and verify the implementation of corresponding safety functions.

The safety standard ISO 26262 suggests design inspection, design walkthroughs, emulation, simulation, and development of prototype hardware as verification techniques for hardware components (cf. [Int09e]). Furthermore, metrics have to be used for hardware components which enable to assess the effectiveness of the technical components to cope with the random hardware faults (cf. Definition 2.13). For example, a quantified FTA can determine the probabilities that hardware components fail (cf. [DT16]). These probabilities have to meet the target values of the safety goals (cf. [Int09e]).

The verification of the lane change assistant is described in more detail in Section 3.6. In addition to the verification, the system analysis impacts the validation of the lane change assistant.

3.4.4.4.2. Validation

Additional to the verification activities also a validation for the integrated system has to be performed (cf. [Int09d]). This validation evaluates the appropriateness of the lane change assistant for its intended use and the adequacy of defined and implemented safety measures. Any unintended activations in a safe situation (*false positive*) and any absented activations in critical situations (*false negative*) have to be excluded for the safety mechanisms of the lane change assistant. Table 3.12 display relation between the classification of situations by the autonomous vehicle systems and the actual criticality of the real situation.

The validation plan defines the time, place, configuration, and execution of all validation activities for the lane change assistant. The validation of the lane change assistant has been performed in the prototype vehicle in tests on public roads and test centers. The validation plan incorporates the configuration of the lane change assistant, necessary equipment, environmental conditions, test procedures, test cases — including driving maneuvers —, and acceptance criteria for these tests.

Table 3.12.: Table for situation classification by autonomous vehicle systems.

		System Classification	
		Critical	Non-Critical
Reality	Critical	True Positive	False Positive
	Non-Critical	False Positive	True Negative

The final impact of the safety analysis is on the safety assessment. In the *safety assessment*, independent assessors have to inspect and evaluate the safety argumentation by car manufacturers for their systems. The impact of the safety analysis on the safety assessment for the lane change assistant is described in the following.

3.4.4.4.3. Safety Assessment

For the lane change assistant, the *safety assessment* has not yet been done because the system is still in development. A safety assessment is commonly performed at near the end of the system development when most of the verification and validation activities have provided sufficient safety-related results. The safety standard ISO 26262 expects a safety assessment for systems and their parts with a classification of ASIL B or higher. The safety assessment must justify that the achieved level of functional safety for these systems is sufficient for the operation of these systems in the real world (cf. [Bir+13]). The safety assessment evaluates all activities and their results in the development of systems and results in the definition of *safety cases*. *Safety cases* document the complete and sufficient consideration of safety requirements throughout all safety-related development activities for the system. This safety argumentation supports the certification process of the systems by national authorities.

Definition 3.14 (Safety Assessment)**.** *In a safety assessment, independent assessors inspect and evaluate the argumentation for the safety of systems (cf. [Bir+13]).*

The following sections give a more detailed description of the implementation, verification, and validation of the lane change assistant.

3.5. Implementation

After the addition of requirements from the safety analysis (cf. Section 3.4) and the revision of the system design, the revised functionality of the lane change assistant and additional safety measures are implemented (cf. Fig. 3.2). Furthermore, hardware components have to be selected and implemented. Finally, all implementation artifacts are verified. Following the system design, the implementation of the lane change assistant

separates itself in the implementation of the functional architecture and the implementation of the technical architecture. Both implementations are described in the following section in more detail.

3.5.1. Implementation of the Functional Architecture

The implementation of the functional architecture focuses on the implementation of the specified functionality for the lane change assistant. Functional components are commonly implemented as *software components* and are implemented only in exceptional case as hardware components, e.g., in field programmable gate arrays (FPGAs). The behavior emerging from software components are validated by engineers using rapid prototyping and verified in software unit tests.

3.5.1.1. Implementation of Software Components

For the implementation of the lane change assistant, the *atomic* functional components of the functional architecture (cf. Section 3.3.1) are refined and implemented as software components. The refinement also applies to the specification of atomic functional components; each requirement of a functional component is refined and allocated to the different software components, which implement the complete or a part of the required functionality. As a result, an architecture of software components is defined for each atomic functional component.
Each *software componento* f the software architecture is implemented in C/C++ source code and consists of several classes with multiple C/C++ functions. Each function implements an algorithm as part of the overall functionality of the functional component. The source code of software components is either manually written by engineers or generated from models, e.g., signal flow graphs (cf. [FPE14]) or statecharts (cf. [Har87]). In the Automotive Domain, commercial tools, e.g., ASCET (cf. [Lef+97]) and MatLab/Simulink (cf. [Bis96]), are used for the model-based implementation of software components. The code generators of these commercial tools allow generating the necessary C/C++ source code from signal flow graphs resp. statecharts. For the deployment of software implementations on ECUs of the technical hardware architecture, the source code of all software components for a system is compiled and linked with necessary libraries as executables (cf. [Ste14]).
The functionality of the lane change assistant is implemented in ADTF. ADTF is a commercial framework for the development and execution of ADAS functions (cf. [Sch07]). The implementation of ADAS in ADTF follows the pipes-and-filters pattern (cf. [Mon+97]). (Sub) functions which are defined in the architecture of the lane change assistant (cf. Fig. 3.18) are implemented as filters. A *filter* in ADTF consists of one ore more C/C++ classes that implemented the functionality of corresponding component and the necessary framework related elements, e.g., ports. In a *filter graph*, all filters for the lane change assistant are connected and with the input and outputs filters for sensors and actuators. At runtime, ADTF provides itself as a runtime environment that manages the control and data flow between filters in the filter graph.

3.5.1.2. Rapid Prototyping

This implementation of the software for the lane change assistant also includes rapid prototyping. Rapid prototyping allows engineers to validate and adapt their algorithms at early stages of the implementation. Algorithms are deployed as source code or models on special rapid prototyping hardware, e.g., the MicroAutoBox[3]. The prototyping hardware is then either deployed in simulations or prototype vehicles. Simulations are the cost and time efficient rapid prototyping environment, while prototype vehicles offer a more realistic validation in the real world. As rapid prototyping is employed from the early stages of the implementation, not all components and functions of the lane change assistant might already be implemented. Therefore, prototype systems offer to simulate the behavior of missing system parts by corresponding models (cf. Section 2.2).
For the lane change assistant, the software implementation in ADTF can be used for rapid prototyping identically in simulations and prototype vehicles. Changes to the implementation are not necessary because ADTF already manages the control and data flow within the lane changes assistant, with other vehicle systems, and with simulation frameworks. The prototype vehicle in the case study was equipped with customer electronic components in order to allow rapid prototyping with ADTF (cf. Section 3.3.2). Nevertheless, rapid prototyping does not address the verification or validation of the lane change assistant regarding its safety assessment. The verification of lane change assistant, commence with unit tests for individual software components. Software unit testing is described in the next section in more detail.

3.5.1.3. Software Unit Tests

A common method for the verification of implementation artifacts in the automotive domain is testing (cf. Section 2.2. Other occasionally used methods include, e.g., static code analysis and code reviews (cf. [Lig09]). In *unit tests*, each software component of the lane change assistant is verified in isolation — independent from other related and connected software components. Unit tests will evaluate if the implemented behavior of classes, functions, and algorithms are consistent with the specification and requirements of the software components. All software components, which interact with the software component under test, are replaced by *mock objects* (cf. [TH02]) or completely omitted from the tests in order to avoid any side effects on the *artifact under test*. *Mock objects* model the external behavior of the mocked implementation artifact. The substitution of related and connect components ensures a consistent interaction for the *artifact under test* with its environment for reproducible tests.
The classes and functions of each software component are verified for a set of test cases. The test cases are derived from the component's specification and requirements. A test case consists of the input data for the *artifact under test*, the response of mock objects in the communication with the artifact, and the expected response (output) of the artifact. All test cases for a *artifact under test* are combined in a *test suite*. In unit tests, artifacts

[3]MicroAutoBox: https://www.dspace.com/de/gmb/home/products/hw/micautob/microautobox2.cfm (accessed: 11/22/2018)

under test are triggered by the input data of the test cases, and a test oracle evaluates the response of the artifacts (cf. [Lig09]). Artifact responses have to match the expected result defined by the test cases. Metrics, e.g., statement coverage or decision coverage (cf. [Lig09]), quantify the coverage of artifact's source code by its test suite. All necessary activities and responsibilities for the unit testing of software components are documented in a *test plan*. This plan includes the definition of the test cases, the setup of the test environment, the execution of the tests, and the evaluation of test results (cf. [Lig09]).
For the lane change assistant, the unit tests of ADTF filter implementations utilize special test filters or the C-unit testing framework[4]. The test filters trigger the *filter under test* via its input ports with manually defined input data. The output of the *filter under test* is evaluated in comparison to predefined expected results for the test case by the test filter. Test filter and the filter under test are integrated into one filter graph isolated from other ADTF filters of the lane change assistant. In this configuration, the filter behavior including the correct implementation of interfaces to the ADTF runtime environment is verified. C-Unit tests do not verify the ADTF filter, but the implementation of algorithms in term of classes and C/C++ functions. For each C/C++ function, a set of test cases is defined. Each test case defines the input values for the *function under test* and the expected response by the function as assertions. In the execution of c-unit test suites, the input values stimulate the function under test, and the assertions assess the function's outputs. Mock objects replace function called within the function under tests in order to enable testing in isolation and avoiding side effects.
Beside the functional architecture also the technical architecture has to be implemented. Car manufacturers do not implement hardware components themselves (cf. Section 3.3.2) but commonly acquire the components from various suppliers. The implementation of the technical architecture for the lane change assistant is described in the following.

3.5.2. Implementation of the Technical Architecture

Besides the functional architecture also the technical architecture has to be implemented. However, hardware components are not constructed and implemented by car manufacturers but are acquired from multiple vendors. These vendors also verify the resilience and robustness of these hardware components. Car manufactures only integrate the third-party hardware components into their execution platform and deploy the execution platform in their vehicles (cf. Section 3.6.3).

3.5.2.1. Procurement of Hardware Components

As described in Section 3.3.2.2, the execution platform for the prototype vehicle uses two customer computer, a gateway, and Ethernet bus in order to enable the efficient development of the lane change assistant. The hardware components, e.g., sensor, actuators, ECUs, and communication buses, used in the technical architecture (cf. Section 3.3.2), are acquired from suppliers and are integrated into the execution platform.

[4]C-unit testing framework: http://cunit.sourceforge.net/ (accessed: 11/22/2018)

The execution of the software components on each ECU and their access to the hardware components, e.g., communication bus or CPU, require the configuration of an operating system (OS) and its services. Similar to hardware components, services of the OS are obtained as off-the-shelf components from suppliers (cf. [SZ13]).
The standard AUTOSAR (cf. [Für+09]) defines an architecture for ECUs with standardized interfaces between application software, e.g., the lane change assistant, and services of the OS — named *basis software* in the standard. The standardized interfaces of the AUTOSAR architecture enable the application software to access basis software components without any knowledge about the possible diverse implementations from various suppliers. A configuration of the operating system for the lane change assistant is not necessary. The customer computers run a standard Linux OS that inherently provides all necessary services for the processing of data and the access to the Ethernet bus.
The result of the safety analysis for the environment perception (cf. Fig. 3.19) require the deployment of a multi-sensor platform for prototype vehicles. Therefore, existing sensors of the prototype vehicle are complemented by additional sensors, e.g., the side and rear RADAR sensors in Fig. 3.19. These additional sensors are installed in the prototype vehicle for the lane change assistant and the highway pilot. The sensors are connected with the execution platform for the lane change assistant via the *gateway* in order to guarantee a redundant 360° view around the prototype vehicle for the lane change assistant (cf Fig. 3.16).
Sensors, actuators, and their corresponding software components often require calibrations. The correct perception of sensors, e.g., stereo cameras, LIDAR, and RADAR sensors, depending on the correct calibration for their mounting location inside the vehicle. For example, a false exit angle of radio waves or LIDAR beams can lead to false reflection points at a position where no object is present resp. to missing reflection points for existing objects. For stereo cameras, the calibration is essential in order to estimate the distances of objects correctly. Car manufacturers have to calibrate sensors themselves or provide sensor suppliers with these parameters in order for the suppliers to calibrate the sensors for them. The same applies to actuators whose ranges of set values have to be calibrated. For example, the possible turn angle has to be calibrated for an electrical steering system.

3.5.2.2. Hardware Unit Tests

Car manufacturers do not perform the verification of hardware components themselves but require suppliers to verify the sustainability and resilience of hardware components in their development process. Failures rates of hardware components have to be sufficiently low and comply with relevant standards, like the safety standard ISO 26262 (cf. [Int09e]). The resilience and robustness of hardware components is commonly evaluated in endurance run using HIL tests or prototype vehicle (cf. Sections 2.2.2 and 2.2.2.7).
However, car manufacturers may randomly test a limited number of provided hardware components in order to ratify the suppliers' statements about the component's sustainability, resilience, and correct calibration. For example, image-based sensor, e.g., stereo

cameras, can be verified in simulations. Traffic situations are generated in simulations and projected onto a screen from which the camera perceives these situations. The identified objects by the camera are compared to the objects used in the simulation. The displayed traffic situations have to be sufficiently realistic for valid verification results.
For the prototype vehicle, no dedicated verification for the sustainability of the computer hardware, gateway, and Ethernet network has been performed. The necessary evidence about the sustainability and resilience of hardware components by legal authorities and safety standards, e.g., the safety standard ISO 26262, only apply to production vehicles but not to the prototype vehicles. The hardware configuration of the prototype vehicle is used for the development, prototyping, and testing of the lane change assistant and is not supposed to be used in production vehicles. However, the original components of the initial production vehicle have already been verified in their development processes and have proven themselves in usage by customers in public traffic.
The software and hardware components, which have already been verified in isolation, have to be integrated in order to complete the system. This integration requires additional verification and validation activities. The following sections describe the right side of the v-model development process (cf. Fig. 3.2) addressing the integration, verification, and validation of the lane change assistant.

3.6. System Integration and Verification

After the implementation and unit testing of all software and hardware components, the development of the lane change assistant proceeds with the integration of these components to larger system parts until the complete lane change assistant is integrated. The integration reverses the decomposition of the lane change assistant in the system design and implementation (cf. Sections 3.3 and 3.5). All compositions of components introduce new functionalities which have not yet been verified by the unit test of the software components (cf. Section 3.5.2). Additional verification is required for composite components in order to guarantee the correctness and safety of the complete lane change assistant. The following section describes the integration and verification of the software for the lane change assistant.

3.6.1. Software Integration

Following the functional architecture (cf. Section 3.3.1), implemented (atomic) software components are integrated with each other to form *composite components* which are further integrated with other components — composite and atomic software components — to form even larger composite components. This integration continues until the lane change assistant and all its functional components, like the *environment perception*, *situation assessment*, and *behavior planning* are implemented (cf. Fig. 3.3). Every *composite component* introduces novel functionalities which emerge from the intercommunication of its internal components. None of the inherent components has exhibited these emerging functionalities in the isolation of the unit tests (cf. Section 3.5.1.3). For example, the

detection and classification of environment objects are only possible because individual components for the object detection and object classification are integrated with the vehicle sensors.
Safety measures defined in the safety analysis (cf. Section 3.4) can be implemented as dedicated software components independently from the original functions of the lane change assistant. In the integration, the components for the safety measures and the component for the original function are integrated into composite components. The composites exhibit the original functionality as long as no faults are present. Safety measure will only intervene with the original functionality if an identified fault from the safety analysis is present.

3.6.2. Software Integration Testing

The emerging behavior of composite components has not yet been verified because the interaction between components has been excluded from unit tests (cf. Section 3.5.1.3). All composites have to be verified in *integration tests* in order to ensure the correctness and safety of emerging functionalities. Similar to unit test (cf. Section 3.5.1.3), sets of test cases verify the behavior of composites components. Test cases define the test input for composite components and the expected behavior for these components based on specifications and requirements.
Integration tests of composite components include the verification of *safety measures* which have been defined for these components in the safety analysis (cf. Section 3.4). The verification of safety measures might require the definition of additional specialized test cases. These tests do not have to verify the correct functionalities of original functions but to verify the detection and mitigation of faults by safety measures. Integration tests of safety measures may incorporate fault injection methods in order to stimulate the emergence of faults (cf. [Ise11]).
For the lane change assistant, the ADTF filter as implementation of software components are integrated in an ADTF filter graph and verified by the simulation framework Virtual Test Drive (VTD) (cf. [Neu14]) in *closed-loop simulations* (cf. Section 2.2). The framework VTD simulates the environment of the vehicle — the virtual world — with its scenery and dynamic objects, e.g., vehicle and pedestrians. The simulation interacts with the ADTF implementation of composite components by their external interfaces. All remaining components of the lane change assistant, e.g., sensors and actuators, have to be simulated by the simulation framework.
The simulation framework VTD allows the verification of the lane change assistant for a set of defined *test cases*. Test cases represent concrete parametrization of a *test scenarios*. A test scenario models the environment by defining the scenery — the static objects of the environment, e.g., road, markings, and sign, as well as the behavior of dynamic objects, e.g., maneuvers of vehicles and paths of pedestrians (cf. Section 7.2.2). A *test case* will be passed if the engineer or an oracle evaluate the behavior of the lane change assistant in the corresponding simulations as compliant with the intended behavior defined by the test case.

The execution of a *test case* in the simulation framework results in a sequence of environmental situations. These situations are used as input for the lane change assistant. Objects from situations are either directly fed into the *scene modeling* as input for the lane change assistant or they are processed by sensor models to incorporate the uncertainty of real sensors into the input for the lane change assistant (cf. Fig. 3.3). The outputs of the lane change assistant are fed back into the simulations in order to update the state of the simulated world and generate new inputs for the lane change assistant in the next processing cycle (cf. Section 2.2). The changes of objects in the simulated world can occur autonomously or as a direct reaction to the output of the lane change assistant.

The overall correctness and safety of the lane change assistant depend on the correctness and safety of its software and hardware. Therefore, the integration of hardware and software has to be verified. The hardware platform must not alter the verified behavior and functionalities of the software. The integration of software and hardware as E/E system and the verification of the system integration are described in the following section.

3.6.3. Integration of the Hardware / Software System

Car manufacturers obtain hardware components from multiple supplier and integrate these components for the E/E architecture of production vehicles (cf. Section 3.5.2.1). The integration includes the connection of ECUs and DCUs to communication systems, e.g., CAN or FlexRay buses, and the power supply of the vehicles.

After the integration and deployment of the E/E architecture, software components are deployed on the ECUs and DCUs. The software deployment includes the definition of timing and scheduling of software components on ECUs and DCUs and of messages on communication buses. For each communication bus, a *communication matrices* defines the content of all messages of the specific bus and their routing between connected ECUs. The scheduling for each ECU defines the execution order and execution duration of deployed software components and their functions.

For the hardware platform of the prototype vehicles, the two customer computers have been installed in the trunks. The computers are connected and the gateway via an Ethernet bus (cf. Fig. 3.16). The gateway connects the computers to the additionally installed sensors and existing communication buses of the prototype vehicles.

The software of the highway pilot is deployed and executed on the two customer computers as ADTF filters in two distributed ADTF instances. The *environment perception* of the lane change assistant is deployed on the first computer while the *situation assessment*, *decision making*, and other ADAS functions are installed on the second computer (cf. Fig. 3.16). The communication between the distributed ADTF instances and with the gateway is implemented by predefined messages in ADTF and organized by the Ethernet controllers and OS services of the two computers without any predefined message scheduling. The gateway connects the lane change assistant with original sensors and actuators of the prototype vehicles by transforming messages of the original communication buses to and from ADTF messages of the Ethernet network.

3.6.4. Verification of the Hardware / Software System

The hardware platform has not been systematically verified because the hardware used in the simulations of the lane change assistant and the prototype vehicle is similar. Knowledge about the hardware setup obtained in simulations of the system verification can be directly transferred to the setup of the prototype vehicles. Nevertheless, the original hardware components of the production vehicles have been verified in the development processes of these vehicles. The only unverified part of this setup is the connection of the gateway to the original vehicle hardware and additional installed sensors. The connectivity of the gateway has been randomly checked at its deployment but has not been systematically verified by a defined set of test cases.

The integration of hardware components, e.g., ECUs, sensor, actuators, and communication buses, is generally verified in HIL tests for production vehicles (cf. Section 2.2.2.4). Hardware components or subsystems are integrated into simulations of their real environments. All connect hardware components are modeled and simulated by the simulation frameworks. For a test of, e.g., an exhaust system, the complete exhaust system is integrated and verified in HIL tests. The control of the exhaust system, as well as the real world, are simulated. The stimuli for each HIL test are defined in test cases along with the expected behavior of the verified hardware components resp. subsystem as acceptance criteria in these tests. The resilience and robustness of hardware components and subsystems can be evaluated in endurance runs (cf. Section 3.5.2.2).

The integration of software and hardware is also verified in HIL tests (cf. Section 2.2.2.4). Therefore, software components are deployed on the corresponding ECUs, sensors, and actuators. Even though the software components have been verified in isolation and as composites (cf. Section 3.6.2), the integration of software and hardware components can reveal faults which have not yet been encountered. These faults are primarily timing and scheduling faults due to the limited resources of ECUs and communication buses. Test cases from software integration tests can be reused to verified the integration of software and hardware (cf. Section 3.6.2). The software-hardware-integration has to exhibit the identical functionality as in MIL and MIL tests of the software (cf. Section 3.6.2).

Tests of fully integrated systems, e.g., the complete highway pilot, are denoted as *system tests*. These *system tests* have to be performed with the highway pilot installed in a prototype vehicle because the lane change assistant and other ADAS of the highway pilot require the perception of the vehicle's real environment. Engineers verify the system on test tracks and public roads and evaluate if the system meets all requirements in the specification of the highway pilot. The complexity of the real world makes it challenging to define comprehensive sets of realistic test cases. Test cases would have to be costly build on test tracks with real vehicles and dummy objects. The *system tests* of the lane change assistant has been solely performed as unsystematic test drives on public roads.

The verification in test drives correlates with the validation of the lane change assistant. The trade-off between verification and validation impacts the efficiency and costs of the system development (cf. [TMJ12; Zha16]). The validation of the lane change assistant is described in the following section.

3.7. Validation in Field Tests

The last development activity for the lane change assistant is its validation (cf. Section 3.1.2). While the verification evaluates if the implementation of components and (sub) systems meet their specifications and requirements, the validation addresses the implementation of the correct and appropriate product. The lane change assistant must satisfy the documented requirements of involved stakeholders — the explicit requirements — as well as the implicit requirements of these stakeholders which have not been considered in the requirements analysis. For example, customers might expect smooth dynamics of lane change maneuvers with low g-forces at higher velocities.

For the *validation*, engineers and stakeholders have to experience the systems in their real environments. In the automotive domain, field operational tests on test tracks and public roads used to be the prevailing method for the validation of vehicles and their systems. In field operaitonal tests on public roads, prototype and production vehicles are driven for million kilometers at different locations in the world (cf. [KW16; Sti13]).

The lane change assistant is deployed for its validation as ADTF filter graph on the customer computers in the trunk of the prototype vehicles. The prototype vehicles are driven autonomously on public highways while engineers supervise and validate the lane change maneuvers for their safety, smoothness, and adequacy. The evaluation results are considered for further optimization and calibration of the lane change assistant in order to align the system behavior with the general preferences of customers.

The engineers also act as safety fallbacks in field operational tests. In case maneuvers by the lane change assistant result in critical and unsafe situations, these engineers have to take over the vehicle control in order to maintain the overall safety.

A common metric for the system quality in V&V is the number of interventionss by engineers and occurring system faults over the total driving distance. The numbers are statistically generalized as the mean time between interventions resp. system faults. The mean time between interventions resp. system faults is used as measures in function safety assessment for the admission of the systems by national authorities (cf. [Int09b]). For example, the California Department of Motor Vehicle oblige car manufacturers to report all interventions (disengagements) for their autonomous prototype vehicles which operate on public roads[5].

The metric has not yet been applied to the lane change assistant because the current development state of the lane change assistant and of the prototype vehicles do not allow for feasible results. The metric will only lead to feasible results if the field operational tests of the validation are performed with large fleets of prototype vehicles for longer distances (cf. [KP16; Win15; WW17]). E/E architectures of these prototype vehicles have to correspond to the final setup production vehicles.

The previous sections have exemplary presented the development of autonomous vehicle systems in the automotive domain — from their requirements analysis to their validation — on the example of a lane change assistant. The following sections analyze the shortcomings

[5] https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/testing/

of this development practice in the foresight of increasing complexity and autonomy for future systems towards highly automatic driving (cf. Fig. 2.4).

3.8. Problem Analysis of the Development Methodology

The lane change assistant has to meet high safety standards in order to enhance the overall road safety and reduce the number of accidents and casualties in public traffic. Even in the presence of system failures, the lane change assistant is expected to operate in the complex environment of the real world safely.
The systematic engineering of the lane change assistant in the presented development process aims to satisfy these high safety standards for the lane change assistant. The development activities (cf. Section 3.1.2) complement each other in order to realize the necessary functionality of the lane change assistant under consideration of potential failures and risks. Nevertheless, safety issues remain for the operation of the lane change assistant in the real world. The main categories of these safety issues are described in more detail in the following sections.

3.8.1. Modeling the System Environment

The system environments have to be explicitly considered in the systems development of the lane change assistant because the environments have a significant impact on the system behavior. The environments of earlier E/E system, e.g., ESC, could be completely specified and modeled because these systems did not autonomously operate in the real world (cf. [Ben+14]). ESC could be developed under the *closed-world assumption* because the system solely interacts with components within its functional domain inside vehicles (cf. Definition 2.4). Values and semantics of signals and message data from other internal components are available at design time. The *closed-world assumption* is not applicable for the lane change assistant because the assistant has to operate in the real world autonomously. The real world continuously changes, and the lane change assistant does not control these changes — they occur autonomously and unpredictable (cf. [FGT11]).
The environment of the lane change assistant — the real world — is considered in its development as *environment models*. This does especially apply to the requirements analysis, safety analysis, and verification of the lane change assistant in simulations (cf. Sections 3.2, 3.4 and 3.6). Engineers and stakeholders implicitly assume a model of the real world for the definition of requirements in the requirements analysis and the identification of risks in safety analysis (cf. Sections 3.2 and 3.4). In the verification (cf. Section 3.6), models of traffic situations and included objects are explicitly defined.
All models have inherent artificial natures and are abstractions of the real world. However, environment models still have to sufficiently represent the real environment of the lane change assistant for feasible results in requirements analysis, safety analysis and verification(cf. Sections 3.2, 3.4 and 3.6). Environment models must not impose more restriction than are present in the real world. All real-world objects which have an impact on the behavior of the lane change assistant, have to to be modeled in sufficient

(a) A model of a Volkswagen Golf MK4.

(b) A real Volkswagen Golf MK4.

Figure 3.20.: Representations of a Volkswagen Golf MK4.

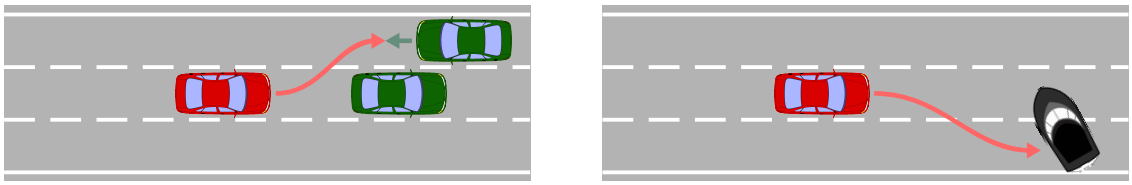
detail. The modeling of these objects includes positions, movements, and characteristics of real-world objects, e.g., cars, pedestrians, roads, or signs. Otherwise, all development results will not be applicable for the later operation of lane change assistant. Insufficient modeling of the environment in requirement analysis and safety analysis will impact all further development activities (cf. Section 3.1.2). Insufficient results from simulations of the verification cannot be assumed as safety evidence for the operation of in the real world.

The number and variations of objects which have an impact on the behavior of the lane change assistant are vast — if not infinite. This enormous complexity of the real world makes it difficult to specify fully and model the environment of the lane change assistant and imposes a significant threat to the safety of the lane change assistant. The lane change assistant has to be developed under the *open-world assumption* (cf. [BNG06]). The *open-world assumption* assumes that possible interactions of the system with its environment in the vast environmental situations cannot be fully considered at design time (cf. Definition 2.4). Any logic for the open-world problem include some assumptions about the real world which reduce its complexity (cf. [JBS13; NM08; Str16]). Traffic situations will remain in which the behavior of the lane change assistant has not been specified nor verified. This underspecification introduces uncertainty about the lane change assistant and its operation in the real world which has to be considered throughout the whole development process (cf. Section 3.1.2).

Real world objects are represented in environment models by corresponding *object models*. Each object model reduces the corresponding real object to a set of defined characteristics. No object model will ever wholly represent all physical characteristics and their small deviations of corresponding real-world objects in all situations. Nevertheless, the object models have to sufficiently represent all characteristic of real-world objects which have an impact on the functionality of the lane change assistant.

For example, vehicles of the type Golf MK4 (cf. Fig. 3.20b[6]) may slightly deviate from each other in their appearances due to scratches and paint quality. The insufficient

[6]taken from: `https://upload.wikimedia.org/wikipedia/commons/f/f7/VW_Golf_IV_front_20071205.jpg` (accessed: 28.6.2017)



(a) Scenario 1: Wrong-way driver on the left lane.

(b) Scenario 2: Motor boat on the road.

Figure 3.21.: Safety critical but irrational traffic scenarios.

consideration of reflection properties for vehicles by sensor models in simulations might result in behavior by the lane change assistant in simulations which diverge from the behavior of the lane change assistant in the real world (cf. Fig. 3.20).

The traffic situations which the lane change assistant may encounter during operation in the real world, are vast — if not infinite — and challenging — if not impossible — for engineers and stakeholders to envisage in the requirements and safety analysis (cf. Sections 3.2 and 3.4) as well as to be verified and validated in the simulations and field operational tests (cf. Section 3.6). Even if we assume, that the system specification has fully captured all possible situations at one point in time, the evolution of the real world will lead to new entities, objects, and situations which have not yet been considered at some point in the future (cf. [FGT11]).

For the lane change assistant, traffic scenarios can be constructed which are less likely to be anticipated in the requirements and safety analysis (cf. Sections 3.2 and 3.4) but which are still critical for the overall safety of the lane change assistant. Both scenarios of Fig. 3.21 describe unusual situations which the lane change assistant might encounter in the real world but are less likely to be anticipated for its environment modeling. In the first scenario (cf. Fig. 3.21a), a wrong-way driver is driving towards the automated vehicle on the left lane. A lane change to the left lane would result in a collision of both vehicles. In the second scenario (cf. Fig. 3.21b), a boat resides on the road. The automated vehicle has to identify this boat and avoid it. For the safe behavior of the lane change assistant in either scenario, the *object models* with appropriate characteristics have to be considered in the development of the lane change assistant. For the second, a boat model has to be explicitly defined and included in the development. For the wrong-way driver, vehicle models have considered the possibility to drive against the intended driving direction. If the directions of vehicles have been restricted in environment models for the lane change assistant to match the way of highways, then the wrong-way driver of Fig. 3.21a) would not have been considered in the development of the lane change assistant.

The following section discusses the shortcomings of the development activities for the lane change assistant from the system point of view. It addresses problems for the environment perception and discusses the decision making and its shortcomings.

3.8.2. Environment Perception and Interpretation

One prerequisite for the correct and safe behavior of the lane change assistant is a reliable the environment perception (cf. Fig. 3.18). The lane change assistant has to correctly perceive the complete vehicle environment at all times — even in the presence of system and sensor faults. The environment perception of ADAS utilizes sensors of various types (cf. Section 3.3.2). Each sensor type perceives the environment by a different physical measuring principle, like e.ge radio waves or light beams. Each physical measuring principle has its particular advantages and shortcomings. For example, the perception range of cameras may be limited by fog while radio waves have difficulties to detect pedestrians correctly (cf. [Pat+17]).
In the safety analysis, the initial sensor setup for the prototype vehicle (cf. Fig. 3.14) has been identified as a *single point of failure* (cf. Section 3.4). It is inevitable for the lane change assistant to integrate multiple different types of sensors into a multi-sensor platform in order to address the shortcomings of different sensor types and their potential failures. The comparison of data from different sensors and the detection of faulty sensor inputs by such a multi-sensor platform enables a reliable perception of the environment in the presence of independent sensor component failures (cf. Section 3.4.4.2).
For the *decision making* of the lane change assistant, data about the environment from different sensors has to be integrated into an internal representation of the environment. This internal environment representation is a model and has the same inadequacies as the models in Section 3.8.1. The internal environment model has to sufficiently represent all relevant object of the real world in a level of detail which corresponds to the functionality of the lane change assistant. Objects and actions which have not been considered in the definition of the internal environment representation cannot be considered in the *decision making* of the lane change assistant.
The physical measurement principle underlying each sensor leads to small deviations (noise) in the sensor data. For example, the propagation times of radio waves slightly vary from the real distances in the world. Algorithms using this sensor data, e.g., *object detection* and *object tracking* may accumulate the uncertainties in the data from multiple sensors within their output data. The processing chain of lane change assistant may accumulate large uncertainties in its data. The uncertainty in data has to be explicitly considered in the *decision making* of the lane change assistant. Small deviations in the environment perception data can lead to unsafe decisions of the lane change assistant because the internal representation represents a different situation for the *decision making* as currently present in the real world.
The verification of the lane change assistant in simulations requires sensors to be substituted by sensor models. Similar to environment models, sensor models are an abstraction of real sensors. The physical measurement principles of sensors are highly complex in order to model them in all detail efficiently. For radio waves and laser beams, processing-intensive techniques like ray tracing (cf. [GGD12]) would have to be incorporated in the simulations. Nevertheless, the sensor models have to represent the processing of real sensors sufficiently including the uncertainty which the real sensors introduce to their data. Sensor models in simulations have to provide sufficiently similar sensor data for

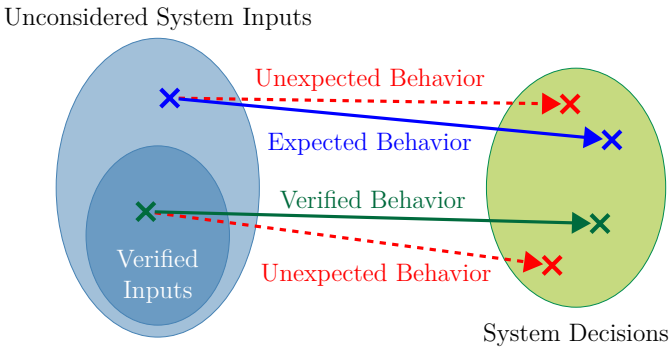


Figure 3.22.: Relationship of system inputs and decision making.

traffic situations as the real sensors would provide in the real world for these situations. Therefore, the performance of sensor models has to be verified and validated with regard to the performance of the real sensors. Otherwise, results from the simulations are not representative of the operation of the lane change assistant in the real world.

Even with perfect environment perception, the behavior of the lane change assistant can be unsafe and faulty. We discuss the shortcomings in the current development practice for the lane change assistant concerning its *decision making* in the following section.

3.8.3. Decision Making in Indefinite Environments

The reliable set of sensors and the redundancy of hardware components, e.g., ECUs, does not guarantee safe operation of the lane change assistant in all situations of the real world. Even if the input from the *environment perception* correctly represents the real world, the *decisions making* of the lane change assistant could still process in appropriate maneuvers for the current environment situation (cf. Fig. 3.18). A reason for wrong decisions in the presence of correct environment representations is the underspecification of the system environment and system functionality for the lane change assistant.

In the development of the lane change assistant, engineers anticipate the situations which the lane change assistant may receive as input in the real world and verify its corresponding decisions (cf. Sections 3.2 and 3.4). Under consideration of the open-world assumption, engineers are unable to specify, verify, and validate the lane change assistant completely in all situations which the lane change assistant may encounter in the real world at design time (cf. Section 3.8.1).

The underspecification of the real world in the development can result in the neglect of side effects which impact on the decision making of the lane change assistant. The internal state of the lane change assistant may diverge between verification in simulations and during operation in the real world due to diverging sequences of prior situations and actions. The minor deviations of physical configurations between different vehicle types and the minor but consistent behavioral deviations of physical components, e.g., sensor, actuators, or ECUs, can have an impact on the behavior of the lane changes assistant.

All these side effects may lead to a nondeterministic system behavior (cf. Fig. 3.22) resulting in maneuver decisions which diverge from the anticipated maneuvers by engineers. This imprecision also applies to situations which have been verified in simulations. Some decisions may be consistent with the expectation of engineers in one situation (cf. blue arrow in Fig. 3.22) while others decisions of the lane change assistant may diverge from the expected system behavior in the same situation (cf. red arrows in Fig. 3.22).
As a result of the underspecification, decisions processed by the lane change assistant in the simulations cannot wholly predict the *decision making* in the real world. In situations, which have not been yet considered in the development of the lane change assistant, the lane change assistant may process incorrect and unsafe maneuvers for correctly perceived traffic situations due to insufficient system functionality. Additional decisions and maneuvers would have to be considered and implemented for the lane change assistant. Additional measures are required which can guarantee the correctness and safety of the lane change assistant during operation in the real world — even for situations which have not yet been considered. These safety measures have to supervise the vehicle and its environment during operation in the real world and identify situations in which the safety of the lane change assistant is compromised. In case the safety of the vehicle is compromised, actions have to be initiated in order to transfer the vehicle into a *safe state* and to mitigate any emerging risks for the vehicle, its passengers, and any object and person in its vicinity (cf. Definition 2.8).
The deployment of a second instance for lane change assistant is not a solution to this problem. Even in the absence of system faults, both instances would always process the same decisions prohibiting any comparison and plausibility checking of these decisions. The redundancy solely enables the detection of random hardware faults but not the detection of systematic faults in the software implementation. Insight from the field of avionics (cf. [BT93; ELS10; LH94]) indicate the necessity for more complex system architectures.
The described problems in the development of the lane change assistant expose that result of the V&V of the lane change assistant are only partially representative for the operation of the lane change assistant in the real world. Situations will remain in which the functionality of the lane change assistant has not yet been verified and validated. In the following section, we discuss the necessity for a metric which quantifies the residual risks for the operation of the lane change assistant in the real world.

3.8.4. Quantification of Correctness and Safety for Acceptability

The general public expects the lane change assistant to behave safe and correct in situations with other automated vehicles, human-operated vehicles, and pedestrians. The lane change assistant must not impose higher risks for the safety of objects and humans in its vicinity than today's human driver [BR15; Fed17]. Car manufacturers are required to provide sound proofs about the positive impact of the autonomous vehicle systems for the acceptance of their autonomous vehicle systems by the general public. The number of saved human lives by these systems must exceed the numbers of lives threatened by them.

National authorities address the public expectation for safe vehicles by requiring car manufacturers to provide a safety assessment of vehicle systems within the certification process of production vehicles. Car manufacturers used to obtain these safety arguments from the statistical quantification of system faults over the mileage in field operational tests. This approach has been suitable for E/E systems under the *closed-world assumption* but is not feasible for the lane change assistant under reasonable costs (cf. [KP16; Win15; WW17]). Following the *open-world assumption*, there will remain situations which have not been encountered for even this huge test distances.
Simulations, field operational tests (cf. Sections 3.6 and 3.7), and formal verification methods (cf. [CW96]) used in the development of the lane change assistant are unable to provide sound proofs about the complete correctness and safety of the lane change assistant. These methods are not able to fully verify and validate the lane change assistant for the vast — if not infinite — number of situations the system will encounter in the real world. Environment models and system models, which are required for formal verification and simulations, are subject to the problems discussed in Section 3.8.1. The required state space of these models would have to be very large — if not infinite — in order to sufficiently represent the real world. Unverified situations and system behavior remain for which the risks of the lane change assistant for objects and persons in its vicinity are unknown.
The infeasibility to provide sufficient proof of the complete correctness and safety of lane change assistant requires the quantification of the residual risks by the lane change assistant during operation in the real world. Otherwise, the positive impact by the lane change assistant cannot be demonstrated to national authorities and the general public. Novel metrics have to be introduced for the quantification of residual risks by autonomous vehicle systems operating in the real world. These metrics have to explicitly consider the infinite space of situations in the real world and the uncertainty in the coverage of real-world situations. For simulations as well as field operational tests, metrics have to determine the coverage of traffic situations by test scenarios and test cases. Such metrics would enable the estimation of the quality of test suites in relation to the covered real-world situations.
The problems in the development of the lane change assistant can be generalized to all autonomous vehicle systems which autonomously operate in the real world. The following section summarizes the problems for the development of autonomous vehicle systems.

3.8.5. Summary of Analysis Results

The previous sections have revealed problems of the current development practice for the development of the lane change assistant. From this analysis, the following shortcomings are derived for the development of all autonomous vehicle systems — especially for higher levels of automation (cf. Fig. 2.4):

- Models used as a representation of the real world in the development of autonomous vehicle systems are always an abstraction of objects, situations, and their characteristics in the real world. Objects and situations remain which are not sufficiently

represented or are not considered at all by the environment models. In the worst case, environment models define limitations for objects and their behaviors which do not correspond to the limitations of corresponding real-world objects. Autonomous vehicle systems are likely to exhibit unsafe behavior in situations with objects, whose behaviors reside beyond the limitation of the environment models.

- The environment perception of ADAS is an interpretation of the real world obtained by sensors. These sensors and corresponding algorithms introduce uncertainty for the values of object and situation parameters in the internal environment representation. This discrepancy may impact the complete functionality of autonomous vehicle systems. The uncertainty of parameters may lead to internal interpretations of the real-world which do not correctly represent the analogous real-world situations. Furthermore, results obtained in simulations will not be applicable for the operation of ADAS in the real world, if models substituting hardware sensors in simulations do not sufficiently represent characteristics of the real sensors.

- The decision making of ADAS may be subject to wrong decisions, even if environmental situations have been correctly perceived. The implementation of the decision making relies on assumptions about the system itself and its environment. Unknown side effects, like deviations of physical components, may have an impact on the *decision making* of autonomous vehicle systems and may lead to unexpected decisions — even for situations which have been verified in simulations. These unexpected decisions may be unsafe and potentially threatening the safety of objects and humans. Diverging and unverified decisions have to be identified and supervised in order to maintain the safety of the autonomous vehicle systems.

- Complete correctness and safety of autonomous vehicle systems cannot be proven in their developments. Residual risks for objects and human remain for the operation of autonomous vehicle systems in the real world. These residual risks have to be determined and quantified in order to gain the acceptance of the general public and national authorities for these systems. The vast — if not infinite — number of real-world situations has to be considered for the qualification of residual risks and the coverage of real-world situations in simulations and field operational tests.

The development of autonomous vehicle systems requires novel engineering approaches which explicitly consider the implication of the real world throughout all development activities. The uncertainty introduced by the interpretation of the real world has to be considered in the design and implementation of the *environment perception* and *decision making*. The V&V of autonomous vehicle systems have to assure realistic modeling of the real world and determine the residual risks of autonomous vehicle systems during operation in the real world as an indicator for their safety and benefits.
All development activities only consider a partial representation of the real world. Future engineering approaches have to incorporate an iterative improvement of autonomous vehicle systems in order to extend the knowledge about the system environment in the development of the novel and previously unknown information about the real world.

The following chapter presents the concept for the improved development of autonomous systems in dynamic, multitudinous environmentss.

4. Emerging Research Questions and Solution Concept

This chapter introduces in Section 4.3 the solution concept of this thesis which addresses the identified shortcomings of the current development process for autonomous vehicle systems in the automotive domain (cf. Section 3.8). Before the solution concept, existing related work is presented in Section 4.1 and the research goal and research questions of this thesis are introduced in Section 4.2.

4.1. Related Work

The analysis in Chapter 3 has disclosed four significant shortcomings for the current development practice for autonomous vehicle systems. Related research for the development, verification, and validation of autonomous vehicle systems is presented in the following. Several authors have identified similar problems for the development of autonomous vehicle systems.

Aniculaesei et al. identify three challenges for the development of autonomous systems (cf. [Ani+18b]); (1) uncertain and unknown environments, (2) adaptive and self-learning systems, (3) open and incomplete artifacts. Their challenges match the identified shortcomings. They also identify the difficulty to completely verify the vast environmental situations of the real world and demand additional safety feature for the operation of autonomous systems in the real world.

The authors of [GRS14; HK16; SSS17; Sti13; WW16] all outline the insufficiency of traditional V&V methods for autonomous vehicle systems and campaign for novel approaches in order to optimize for the safety assurance of autonomous vehicle systems. The mileage, which is required to verify the safety of autonomous vehicle systems in real-world tests, is not feasible for car manufacturers under reasonable time and costs.

Most current engineering approaches for autonomous vehicle systems embrace the improvement of computing performance, high-definition world maps, or high-quality sensor perception. The manifest of all these improvements in the near or mead-term future might be plausible but has a low probability (cf. [SSS17]). Shalev-Shwartz et al. call for an abstract domain interpretation of road traffic in the safety assurance of autonomous vehicle systems.

Weitzel and Geyer address in [Wei+14] the representativity of system requirements in the evolution of autonomous vehicle systems. Comprehensive statistical data with an arbitrary level of detail would be required for system requirements which precisely represent the real world.

A significant focus of current research for autonomous vehicle systems is on V&V techniques which ensure high standards of safety for the operation of autonomous vehicle systems in public traffic.
An early idea of the presented V&V approach in this thesis (cf. Section 4.3) has been first presented in [MRS14] and further refined in [MHR15; MHR16]. [MHR15] introduces the architecture of the runtime monitoring (cf. Section 4.3.1). [MHR16] presents the basic concepts of the runtime monitor engineering in a case study on an industrial ADAS — a lane change assistant. Chapter 6 and Chapter 8 describe the engineering of runtime monitors resp. the setup and results of the evaluation on the lane change assistant in more detail.
The following section gives an overview of other related V&V approaches for autonomous vehicle systems.

4.1.1. Testing of Autonomous Vehicle Systems

The verification and validation have a crucial role in the development of safe autonomous vehicles. Tests are performed in the virtual environments of simulations or the real world within field operational tests. A taxonomy for testing approaches of ADAS and autonomous vehicles are given in [Ste+15]. Stellet et al. investigate testing approaches for four methodological challenges; test criteria, test scenarios, metric, and ground truth reference.

4.1.1.1. Real World Testing

Field operational tests are an integral activity in the verification and validation of autonomous vehicle systems (cf. [Bar+16; BC10]). Even though, the verification and validation solely in field operational tests are not feasible for autonomous vehicle systems field operational tests are still beneficial for the development of autonomous vehicle systems due to their inherent realism (cf. [KW16]).
Currently, mainly statistical quality assurance techniques are used to guarantee the safety of vehicle systems. The vehicle systems are observed along large numbers of driven test miles. The safety of these systems is judged based on the rate of faults during these test miles [SZ13; Wei+14]. Nevertheless, real-world tests still offer possibilities for efficiency improvements. Glauner et al. analyze relevant events in field operational tests and aggregate performance coefficient for each road in these tests. These coefficients depict the impact of specific roads in these tests and enable the planning of more efficient field operation tests (cf. [GBH12]).
Critical traffic situations such as collisions are hard to evaluate in field operational tests without high risks for drivers and vehicles. Therefore, researchers have incorporated simulations techniques into real world tests. Bock et al. use augmented reality for the injection of virtual obstacles into the fields of view of human drivers (cf. [Boc09]). However, this approach is not applicable to autonomous vehicle systems because the driver is substitute by the autonomous vehicle system. Sefati et al. introduce virtual objects into the object recognition of automated driving functions in [SSW13] in order to

verify an emergency evasion functionality. Evasive maneuvers are identically performed in the simulation and the real world. Both approaches do no benefit from the performance and reproducibility of simulations.
The virtual assessment of automation in field operation (VAAFO) approach in [WW15] can be seen an evolution of the approach by Sefati et al. Automation systems are deployed alongside the VAAFO system in an *open-loop* in real vehicles (cf. Definition 2.21). However, the automation systems do not interfere with vehicles' real behavior. Human drivers still control the vehicles. The automation system drives the vehicle in a simulation of the real world. The simulations are generated by the VAAFO based on an additional real sensor perception system. The trajectories of the automation systems in the virtual worlds are compared to the trajectories of the real vehicles in the real world.
The VAAFO approach is similar to the *shadow mode* by Tesla Motors (cf. [Kal17]). Tesla Motors equips their production vehicles with new versions of their autopilot in *shadow mode*. The autopilot system reacts to the data from the real vehicle sensors, but its actions are not considered for the vehicle control. The behavior of autopilot systems is evaluated in traffic situations in comparison to the drivers' actions and recorded for later analysis and improvements. A simulation of the vehicle's environment, like the one in the VAAFO approach, is not incorporated in the *shadow mode*. The recording of data about the vehicles and their environments by the *shadow mode* is similar to the recording of system behavior and traffic situations during operation in the real world in this thesis (cf. Section 4.3).
The extensions of field-operational tests by all these approaches do not significantly address the requirement to drive millions to billions of test miles in order to soundly verify the objects or extension to production vehicles. Prototype vehicles with the VAAFO system still have to accumulate this enormous mileage on public roads. An alternative approach to the verification and validation of autonomous vehicle systems is simulation-based testing.

4.1.1.2. Simulation-based Testing

Simulation-based testing is currently the most promising approach for an efficient V&V of autonomous vehicle systems. However, simulations will not entirely replace field operation tests. This raises the question for the ratio of simulation-based tests and field operational tests in the verification and validation of autonomous vehicle systems. In [Böd+18], Böde et al. probabilistically calculate an optimal trade-off between simulation-based tests and real-world tests. The trade-off considers the individual costs for simulations and field operational tests, the accuracy of virtual models for the prediction of real-world behavior, and the compliance of the autonomous vehicle systems to their requirements.
Several frameworks have been introduced for the simulations of autonomous vehicle systems and their environments. The off-line simulation tool PRE-crash Scenario ANalyzer (PRESCAN) offers an integrated solution for reliable simulation of intelligent vehicles, their dynamics, sensors, and environments based on multi-agent real-time simulation [Gie+04]. After successfully passing PRESCAN, the test scenarios can be used in VEHIL tests (cf.[HPT10]).

Neumann-Cosel et al. present in [NDW09] the modular simulation platform Virtual Test Drive (VTD) for the simulation of vehicle environments, their infrastructure objects and other dynamic objects (cf. Section 2.2.1). Parameterized sensor models gather information about the virtual vehicle environment. The information about the vehicle environment is processed by software models of the autonomies vehicle systems, electronic control units, or driving simulators [Neu14]. The VTD simulation system can be built into cars to access the vehicle's powertrain, steering, and communication channels [Las+10].

Gruyer et al. propose the simulation environment Pro_SiVIC in [Gru+10]. Pro_SiViC focuses on physical sensor capabilities but also supports vehicle dynamics, traffic generation, and driving assistance. The simulator offers a graphical environment and can be coupled with other tools for the usage of their control algorithms.

All these simulation frameworks can be incorporated into various XIL tests in order to improve time and costs of testing autonomous vehicle systems (cf. Section 2.2.2)

In [Ulb+16], Ulbrich et al. discuss and compare advantages and limitations of in situation-based open-loop tests and scenario-based closed-loop tests for testing and validating algorithms of the tactical behavior planning in a lane change assistant.

In this thesis, the simulation environment VTD is used in the case study on a lane change assistant because it has had already been applied in ongoing research of the project partners (cf. Chapter 8). Nevertheless, other simulation environments could be used within the holistic engineering approach(cf. Section 4.3). Standard file formats, e.g., OpenSCENARIO (cf. [Dup16]), enable simulations in all supporting simulation frameworks by defining a standard specification of scenarios, sceneries, and dynamic objects.

For the verification and validation of sensors and the analysis of their impact on vehicle functions, sensors are modeled in simulations in various degrees of precision — ranging from perfect sensor models to highly realistic models with jitter in their sensor data.

In [PGV12], Pechberti et al. present their sensor model for the RADAR simulation in Pro_SiVIC. The model reproduces the physical characteristics and uncertainties of the antenna hardware as well as the propagation of radio waves in the real world and provides the simulations with realistic sensor data about the simulated virtual worlds. The RADAR model allows analyzing the robustness of autonomous vehicle systems for realistic RADAR sensor data in simulations.

All physical sensor models, e.g., LIDAR, RADAR, cameras, and ultrasonic sensors have to be validated for their accurate representation of the real world in simulations. The approaches for validation of perception sensor models by [Rot+11; Sch+17] use test drives in the real world to gather realistic sensor data. The real world data from the real sensors is compared with the synthetical data of the sensor models in equivalent simulations. Schaermann et al. extend the validation approach of Roth et al. by explicitly considering the raw data and object list data from sensors and sensor models in the evaluation.

Gruyer et al. validate their camera sensor models for Pro_SiVIC in [GGD12] based on two types of test targets: dot charts and retro-lighting charts. The camera models are designed for varying real-world camera system using their real-world calibration data.

Images as outputs of the camera sensor models are recorded and compared to images of the real camera systems for the validation.
In [Ver+00; VVP02] HIL tests are extended to a full VEHIL tests allowing the incorporating of the real vehicle sensors, actuators, and vehicle dynamics into simulations. The environment of the vehicle under test (VUT) is simulated for test scenarios as relative motions of other traffic participants are imitated in the real world by dedicated hardware objects relative to the VUT on the test bench.
The impact of simulation-based testing for autonomous vehicle systems is subject to the quality of modeling the vast number of real-world situations in test scenarios and test case (cf. Section 3.8.1). Approaches for the generation of test scenarios and test cases can be distinguished into mathematical/combinatoric approaches and real-world data approaches (cf. [Aga+16]).
Simulation-based tests are a promising approach to the accumulation of millions to billions of test miles for the safety assurance of autonomous vehicle systems (cf. [KP16; Sti13; WW16]). However, the impacts of simulation-based tests depend on the scope and realism of simulated test scenarios and test cases.
The VEHIL approach seems to be especially promising in addressing the significant shortcomings of simulations and real-world tests — insufficient realism of simulations and costs and time of real-world tests. The VEHIL approach is very close to real-world testing by incorporating the real physics of real sensors and actuators. The complete processing chain of autonomous vehicle systems with sensors and actuators can be verified in VEHIL tests. However, the impact of the VEHIL approach is still subject to the scope and realism of test scenarios and test case. The VUT is only verified for a defined set of test scenarios and test cases, but their realism is not validated. An approach like the one in this thesis can help to gather real-world data for the definition of realistic test scenarios and test cases.
The following sections elaborate on methods for the generation of test scenarios and test cases.

4.1.1.2.1. Mathematical Test Case Generation

One approach for the generation of test cases is the use of mathematical and combinatoric techniques to define new test cases for autonomous vehicle systems systematically. Inputs and outputs of autonomous vehicle systems are analyzed and combinatorially combined as test cases which cover a large quantity of the problem space for autonomous vehicle systems.
Schuldt et al. present in [Sch+13] a modularized virtual test tooling kit for the generic generation of test scenarios for simulation-based verification. The tooling kit outlines the process from the verified ADAS over the identification and analysis of relevant parameters, definition of test scenarios, verification in XIL tests, and evaluation of test results by metrics. The tooling kit further defines a four-layer model for the flexible combination of the underlying road topologies, obstacles, traffic situations, and weather conditions in test scenarios.

In [Ber10], Berger models test scenarios from system requirements and metrics from customers' acceptance criteria for automatic acceptance testing of autonomous vehicle systems in simulations. The test scenarios are defined in a domain specific language (DSL) called *ScenarioDSL*.
Some approaches generate tests scenarios by classifying traffic situations. Saust et al. define in [Sau+09] an approach for the test case generation based on the classification of critical traffic situations from crash data databases. Sippl et al. derive test cases from probabilistic environment-sensitive behavior simulations data (cf [Sip+16]). The simulation data is filtered for relevant situations in which other traffic participants impact the behavior of the automated ego vehicle. The remaining situations are rated based on developer specified factors and stored in a situation catalog. Test scenarios for are defined in a textual DSL from the situations in this situation catalog.
Berger et al. address in [Ber+14; Ber+15] the systematic generation of EuroNCAP conform test scenarios[1] for simulation-based verification of active safety systems under the consideration of tolerance ranges for a specific system and environment parameters. Possible variances of parameters over time are modeled as a graph. Trajectories are derived from the graph as test cases for the simulations. The authors enhance the evaluation of test results by introducing tolerance ranges in [Ber+14].
The authors of [Gäf+08; Tat15; TMJ12] present a tool-based generation of test cases for simulation-based testing of autonomous vehicle systems. The tool *TestWeaver* uses intelligent search in order to automatically analyze and classify the behavior of autonomous vehicle systems in simulations for the automatic generation of different test scenarios. *TestWeaver* controls specified parameters and inputs of the system and monitors the system behavior in simulations on system requirements and quality criteria.
Khastgir et al. present in [Kha+17] an automated constrained randomized approach for the definition test scenarios and test cases using the tool Vitaq[2]. The approach applies randomization to test scenarios and test cases for the variation of vehicle trajectories, environment, and traffic in the simulations by intervening the real-time communication between simulation and SUT. Constrained randomization enables the intelligent exploration of the problem space in simulations in order to find the corner cases for which an ADAS and automated systems are likely to fail.
Aniculaesei et al. present in [Ani+18a] an automatic requirements-based test-case generation for an adaptive cruise control system. Natural language requirements are formalized in LTL. The LTL formulas are negated as trap properties under consideration of code coverage metrics. A formal system model is used by the model checking tool NuSMV to generate counterexamples for the trap properties (cf. [Cim+02]). The counterexamples represent traces through the system model which allow the derivation of test cases. The test cases verify the system requirement for different mutants of the initial system.
The authors of [Bau+07; Bau+08; Sie+11] introduce usage models for the risk- and model-based testing of software-based (embedded) systems. Annotated UML diagrams from the requirements analysis enable the definition of state-based usage models. These

[1] https://www.euroncap.com/en/for-engineers/protocols/ (Accessed 12/05/2018)
[2] http://www.vertizan.com/methodology; (Accessed: 10/03/2018)

usage models represent the risk profile of each SUT. Transitions in the usage models are annotated with the corresponding risks. Test cases for the SUT are derived from valid paths through these usage models. For risk-based testing, generated test cases are prioritized in risk-based test plans based on the accumulated risks by all transitions in each test case's path.
In [Oli+16], Olivares et al. use Markov Chain and Markov Chain Monte Carlo methods to generate road topologies for test scenarios. Essential road parameters, e.g., geometry, type, and the number of lanes are deduced from OpenStreetMap (cf. [HW08]). All relevant road parameters are repented in a stochastic model by probability density functions and conditional probabilities. This stochastic model allows generating test scenario with critical combinations of road parameters which have a high possibility to reveal unsafe system behavior.
Zhao et al. introduce in [Zha16] an approach to accelerate the evaluation of automated vehicles by eliminating repeating and uncritical parts in naturalistic driving data for simulations. Stochastic models for the behavior of traffic participants are derived from real-world naturalistic driving data and optimized by reducing its non-safety-critical portion. Monte Carlo simulations (cf. [Moo97]) use the optimized models in order to evaluate interactions between the automated ego vehicle and other traffic participants with higher criticality. Results from the Monte Carlo simulations help to understand the performance of the automated ego vehicle under naturalistic driving conditions. In [HLZ17], Huang et al. incorporate the *Kriging model* as statistic models for the behavior of traffic participants.
Mathematical and combinatorial approaches for the generation of test scenarios and test cases in simulation-based tests are used to cover large scopes of the input/output space of systems whose input space has been completely specified. Autonomous vehicle systems incorporate inputs with complex data, e.g., object-oriented description of the systems' environments, that makes it difficult — if not impossible — to fully cover the input/output space of autonomous vehicle systems by mathematical and combinatorial techniques. It remains questionable if mathematical and combinatorial techniques able to generate test scenarios and test cases that sufficiently model relevant and critical real-world traffic situations.
Nevertheless, general aspects of these approaches, e.g., the layered model for scenarios in [Sch+13], can be incorporated by more-elaborated approaches. This thesis considers the layered model for test scenarios in [Sch+13] for the definition of test scenarios (cf. Section 7.2.3).

4.1.1.2.2. Test Case Generation from Real World Data

Complementary approaches for the generation of test scenarios and test cases incorporate data from real-world test drives.
The test scenario and test case generation by mathematical and combinatoric approaches can verify the behavior of autonomous vehicle systems in a vast variation of traffic situations, but some modeled traffic situations might not be representative for the operation of the autonomous vehicle system in the real world or be completely unrealistic.

Other approaches incorporate data from real-world driving for the generation of test scenarios and test cases.

In [PHK17], critical situations are identified in data from naturalistic driving studies based on their occurrences in crash databases. The critical situations are stored with their parameters in a situation catalog. Scenarios are defined from the situations in the situation catalog for the qualitative and quantitative evaluation of vehicle safety functions.

Lages et al. briefly describe in [LSK13] the definition of test scenarios from real-world data of reference sensor system. Zofka et al. use in [Zof+15] real-world data for the creation of critical traffic scenarios. The real world data is recorded in test drives by a reference sensor system. The road layouts and trajectories of traffic participants from the real world test drives are recreated in the test scenarios. Spatial and temporal modifications of the recorded trajectories enable the creation of additional test scenarios. The approach is limited by the fixed, derived, and parametrized vehicle maneuvers and allows only for open-loop testing (cf. Definition 2.21).

Peters et al. analyze in [PHR16] recordings from real-world test drives to represent the specific states and transitions in these test drives as automaton models. Segments of consistent behavior are identified in the recorded data and classified to known segments in a knowledge base. The sequence of classified segments is transformed into an automaton model. Classified segments represent the states of the automaton model. Transitions between classified segments in the test drives are mapped to transition in the automaton model. The model is annotated with additional information about the number of instances for each segment and each transition as well as the average duration of each segment.

Bach et al. describe in [Bac+15; Bac+17a; Bac+17b] a reactive replay approach for utilizing recorded test data during in the virtual verification of ADAS. The reactive replay approach identifies casual dependencies of inputs for ADAS in order to identify system inputs which can be represented by conventional plant model and system inputs which have to be stimulated by data from conditioned data records. Coherent and reactive stimuli for the SUT are enabled by changing the inadequate domains of recorded data to match the input domains of the plant model.

Driving scenarios are specified by a domain model which abstracts from the real world and characterizes temporal and spatial information on a logical level in an omniscient view composed of sequential different acts and maneuvers (cf. [BOS16]). Graph-based rule models ensure the consistency of specified scenarios.

In [Bac+17c], Bach at al. present a two-step approach for the selection of test scenarios in verification of ADAS with the reactive replay approach. The two-step approach consists of a specification-based classification and a data-driven reduction. Available real-world scenarios are categorized based on criteria and properties from system-level requirements for an initial scenario selection. The initial set of scenarios is further reduced based on their cross-parameter coverage of system inputs and outputs to minimal scenario subsets with significant diversity under avoidance of repetitive situations. The usage of the reactive replay approach limits the combinations of parameters in the selection of scenarios to recorded real-world driving data.

In [Luc+16], Lucchetti et al. automatically identify most common driving scenarios in data from on-road experiments based on spatial relations of vehicles in relation to the ego-vehicle in recorded situations.
The approaches in [Bac+15; Bac+17a; Luc+16; Zof+15] are similar to the engineering approach presented in this thesis (cf. Section 4.3). The approaches use data from real-world drives for the definition of simulation-based tests. However, none of these approaches validates the verification results from simulations in the real world.
The approaches of [Bac+15; Bac+17a; Zof+15] vary recorded system parameters for simulations and do not create an abstract representation of the system state and environment state for universal usage in various simulation frameworks. The approach of [Luc+16] incorporates a description of the environment similar to the environment description for the lane change assistant in the case study (cf. Chapter 8). They require the modeling of reference scenarios before the analysis for their identification in real-world test drives. Their approach does not address an iterative extension of these reference scenarios.
Simulation-based tests are an efficient approach to verify and validate autonomous vehicle systems in many variations of traffic situations under reasonable costs. However, it is unlikely that simulation-based tests will verify autonomous vehicle systems and their decision making for all possible real-world situations (cf. Section 3.8.3). The following section presents related work for the verification of autonomous vehicle systems during operation.

4.1.2. Vehicle Diagnosis and Runtime Monitoring

Simulation-based testing is unlikely to verify all real-world traffic situations. The behavior of autonomous vehicle systems during operation has to be monitored and supervised on in addition to the verification in the system development and in order to identify unsafe system behavior and mitigate emerging safety risks. The following sections present related work in the field of (on-board) diagnosis and runtime monitoring for the verification of autonomous vehicle systems during operation.

4.1.2.1. On-board diagnosis

Onboard diagnosis has been widely used in the automotive domain for monitoring the correctness and reliability of physical components. The field of onboard diagnostics mainly uses supervision, fault detection, and fault management techniques based on single physical system parameters and mathematical models of the system's physical behavior in order to detect, record, and resolve deviations and faults in the controller's behavior (cf. [Ise05; Ise11; Ise97; Mar+13; MFG11; SZ13; Wei+14]). The recording of deviations and faults enables later off-board diagnosis in, e.g., workshops (cf. [SZ13]). The impact of faulty system behavior has to be mitigated by, e.g., deactivation of faulty components or reduced functionality until the deviations and faults are resolved in, e.g., workshops.
Redundancy is a common approach for fault-tolerant sensor systems and control systems in the automotive domain. It is commonly distinguished into physical, dynamic, and

analytical redundancy (cf. [Wei+14]). For physical reduced multiple identical or diverse components are deployed for the identical system functionality. A *voter* validates the output signals of all components and select the correct output signals for the remaining system components based on the majority of identical output signals. Dynamic redundancy employs multiple components. In addition to the original system component, a standby-component is deployed. The standby-component remains inactive until the original system component produces faults. In the presence of faults by the nominal component, the standby-component takes over the processing, and the original component is deactivated. Analytical redundancy incorporates analytical models to calculate a part of the complete measurement signals of the original system component. A comparison of signals by the original components and the analytical models by, e.g., correlation analysis, spectral analysis, or wavelet analysis enable the identification of component faults (cf. [Wei+14]).
The sensor perception of autonomous vehicles is commonly designed that perception components validate each other. Failure models are incorporate to evaluate and determine the quality of sensor data (cf. [Wei+14]). This information can be considered by components of the situation assessment and behavior planning for more accurate and safe maneuver processing (cf. Fig. 3.3).
The standardized E-Gas monitoring concept [EGA13] is widely used in the automotive industry for the diagnosis of EGAS systems. The E-Gas monitoring concept has three layers; layer 1 refers to the standard engine control function, layer two refers to the functional monitoring, and the third layer is responsible for the controller monitoring.
The functional monitoring compares the actual engine torque with a calculated "permissible engine-torque" and the current acceleration with a separately calculated "permissible vehicle acceleration". The "permissible engine-torque" is calculated based on the setpoint value of the gas pedal and any external torque requests while the "permissible vehicle acceleration" is processed from the gas pedal set point, torque requests, current vehicle speed, and engine speed. The controller monitoring supervises the hardware which executes the EGAS functionality. Functional and controller monitoring initiate a fault reaction in the presence of a system fault in order to maintain a safe operating state.
The monitoring of systems' internal processing by onboard diagnostic systems is not sufficient for autonomous vehicle systems. Onboard diagnostic systems are not sufficiently considering the complex external traffic situations in which autonomous vehicle systems have to operate. Even in the absence of component faults, the nominal functionality of autonomous vehicle systems may result in unsafe system behavior.

4.1.2.2. Runtime Monitoring

Runtime monitoring approaches have to monitor the complete behavior of autonomous vehicle systems in the fast variations of traffic situations. Runtime monitors distinguish themselves in their description of the monitored properties and their integration with the monitored system (cf. [Hav11]). A good overview of basic runtime monitoring approaches is given in [GP10]. The following section presents runtime monitoring approaches which are related to the runtime monitoring concept in this thesis.

4.1.2.2.1. Monitoring Architecture

The following related work addresses the integration of runtime monitors and monitored system. In [Rd04], Ricardo et al. present a runtime monitoring framework for systems with identical functionality but slightly varying implementations operating on similar hardware. The runtime monitoring framework equally monitors system functionality and hardware while hardware information collectors monitor individual hardware components. The system functionality is intrusively monitored by events from so-called sensors in the system implementation. A runtime checker evaluates based on the events and information about the hardware if the system execution matches the system requirements.

The publications [Ack+08; Ray+09] assess the instrumentation-based verification in the design of a body electronics application. Monitor models are integrated into system models in Matlab/Simulink (cf. [Bis96]) for the runtime monitoring of the modeled system functionality. The monitor models monitor the data flow in the system models in order to detect violations of the requirements. Each requirement is encoded by its monitor model. For the evaluation of system models, test data is automatically generated based on predefined test coverage criteria.

The authors of [Ors+02] use software tomography for a continuous, minimally intrusive runtime monitoring. They divide the monitoring task into smaller monitoring task and assign these small monitor subtasks to different instances of the monitored systems in order to reduce the side-effects from the instrumentation on the performance of the system. The partial information from the subtask is integrated and used for modifying and updating the system.

All previous approaches use code instrumentation for access to information about the execution of the monitored systems. Code instrumentation is not suitable for autonomous vehicle systems in general because the code changes may impact on the system functionality on the embedded hardware components. The versions of autonomous vehicle systems verified and validated in the development should match the versions in production vehicle for customers in order for the verification results applying to the operation of these systems on public roads. For the engineering approach in this thesis, code instrumentation is disqualified because the same version of autonomous vehicle systems must be used in the simulations of the system verification and during operation in the real world. Runtime monitoring approaches with non-intrusive access to system data are presented in the following.

Nelissen et al. propose in [NPP15] a new reference architecture for inline runtime monitoring using buffers for monitored system events. These buffers decouple the runtime monitor and the monitored system that the monitored system must not have any knowledge about the runtime monitoring. Relevant events of the system are stored in individual buffers for each event. The runtime monitor is periodically executed independently from the monitored system and synchronously accessed by the runtime monitoring for the evaluation of the system behavior.

Nguyen et al. present in [Jak+15; Ngu+16] a framework for assertion-based monitoring of automotive systems-of-systems with mixed criticality. The framework uses qualitative and quantitative interpretations of signal temporal logic (STL) assertions over mixed-

analog signals to automatically synthesized hardware monitors. These monitors can be implemented directly from hardware description language (HDL) code in FPGA hardware.
In [Kan+15; Kan15; KFK14], Kane et al. use a passive network monitor to monitor high-level system properties from the observed network state for autonomous vehicles. The network state is acquired by passive observation of the system's broadcast buses and mapped to propositions of discrete temporal logic for the evaluation of system properties about the correct and safe system behavior. The usage of hardware monitoring has the benefit of non-intrusive access to data of the monitored systems.

4.1.2.3. Runtime Monitoring Properties

Runtime monitoring approaches commonly require the definition of system properties about the correct system behavior. Most runtime monitoring approaches define safety and liveness properties in formal logic or model correct resp. incorrect reference system behavior within automata.
Kane et al. map observation of the system's broadcast buses to propositions of discrete time metric temporal logic (MTL) (cf. [Kan+15; Kan15; KFK14]). These propositions are derived from relevant safety requirements and invariants for autonomous vehicles using specification design patterns and semi-formal techniques. The eager algorithm of the runtime monitoring periodically evaluates the requirements and invariants based on the propositions each and enables the earlier detection of property violations providing the system more time to attempt a recovery.
Cimatti et al. present in [Cim+08; Cim+10] their property-based methodology and techniques for the formalization and validation of high-level requirements for safety-critical applications. Requirements are specified in first-order temporal logic after an informal analysis. Consistency and entailment of required properties, as well as the possibility to model desirable scenarios, are evaluated for the formalized requirements by automatic but incomplete checks using model checking techniques
Schamai uses in [Sch13] simulation models for the design verification. The simulation models include formalized natural-language requirements and allow to monitor requirement violations by requirement violation monitors in simulations of design models with OpenModelica (cf. [Fri+06]). Scenarios models create test inputs for the design models in the simulations. The language ModelicaML is introduced for the formalization of requirements by the simulation models using UML and Modelica (cf. [FE98]).
Kneer et al. propose in [KK15] an approach for generation of requirements monitors in self-adaptive systems from a requirements specification (cf. [Bru+09]). The requirements monitor consists of a rule engine and an impact analyzer which process a suitable configuration of the self-adaptive system in changing situations. The rule engine monitors assertions based on runtime data from probes in the self-adaptive systems. The impact analyzer estimates the impact of broken rules using a goal model. The system requirements are defined in a semi-formal requirements language for the generation of probes, assertions, and the goal model.

In [HMF14], Heffernan and et al. use the ISO 26262 for the definition of monitored properties for automotive embedded systems. Functional safety requirements from the safety analysis of ISO 26262 are expressed as formulae in past time LTL and evaluated at runtime by a non-invasive runtime monitor listing on microcomputer's system bus. Past time LTL is a variation of the LTL (cf. [Pnu77]).
Reinbacher et al. also use in [Rei+11] past time LTL to monitor system requirements but they implement an intrusive runtime monitor by instrumenting the application code.
The authors of [Ani+16] use model checking to verify assumptions of autonomous systems about their environments at development and runtime monitors to supervise these assumptions during operation in the real world. Timed automata model the behavior of systems and their environments. System safety properties are defined as a formula in timed computation tree logic (TCTL). The correctness of these models concerning this property is verified at design time. Runtime monitors are derived from the models for supervising the safety properties during operation in the real world.
Nenzi et al. introduce a topological spatial surround operator for signal spatio-temporal logic (SSTL) with a qualitative and quantitative semantics. Monitoring of spatial modalities requires different monitoring algorithms than for timed modalities. Efficient runtime monitoring algorithms are introduced for monitoring the spatial-temporal behavior by complex systems with qualitative and quantitative semantics. For the quantitative semantic of the surround operator, a novel fixed-point algorithm is introduced.
Machin et al. describe in [Mac+14; Mac+18] the runtime monitoring of autonomous systems based om safety rules. Safety rules are automatically synthesized based on the concept of safety margins. Safety rules describe system behavior in the presence of potential hazards by defining necessary interventions for violation of safety invariants. Safety invariants are formally expressed as predicates over system variables from the margin analysis. The margin analysis splits domains of the system variables into discretized intervals or sets. All safety rules for a safety invariant are accumulated into a safety strategy. A safety monitor implements all safety strategies for a system. The safety monitor evaluates and ensures the safety of the autonomous systems during operation.
Krüger et al. use [KMM07] aspect-oriented development techniques for runtime monitoring of distributed systems. They model the communication of distributed system components in message sequence charts (MSCs) for the generation of runtime monitors. The code of the distributed system is instrumented for the runtime data by using aspect-oriented techniques.
Ahluwalia et al. present a similar model-based approach for the runtime monitoring of end-to-end quality of service (QoS) properties in [Ahl+05]. Interactions between system functions and corresponding QoS properties — sets of real-time constraints — are modeled in extended MSCs. The QoS properties are deployed to all system components and monitored by a so-called component monitor (CMonitor) based on the real-time constraints.
The authors of [Fea+98] use runtime monitoring of system requirements for the selection and initiation of self-adaption tactics. The system requirements are defined by the KAOS goal-driven specification methodology (cf. [VDM95]). Goals are formally defined as assertions in real-temporal logic (cf. [Koy92]). These assertions are mapped to sequences

of events in the FLEA monitoring system (cf. [Coh+97]) and evaluated for events from the monitored system.
Another approach using goal-driven specification for runtime monitoring is the approach in [Wan+07]. Requirements are defined as annotated goal model and transformed into propositional formulas for their evaluation in the propositional satisfiability (SAT) solvers SAT4J (cf. [LP10]). SAT4J evaluates the propositional formulas offline on program execution traces. The program execution traces are gathered by instrumenting the code of the system with probes and stored in compact propositional encoding.
All runtime monitoring approaches evaluate the system behavior during operation, but none of the presented publications integrates their runtime monitoring approach in the context of the system development processes. The majority of approaches manually derives the monitored system properties from system requirements or defines these properties on the signals of the monitored system. As a result, the system behavior is monitored on the system level without any abstraction. Therefore, the investigation of emerging faults and insufficient system behavior for the improvement of systems requires system experts.
The work in [Kan+15; Kan15; KFK14] incorporates a similar pattern-based analysis for the definition of the monitored properties from system requirements. However, the properties are defined in propositional logic. Proportional logic is insufficient for reasoning about the environments of autonomous vehicle systems because propositional logic cannot reason over objects in the environment (cf [HR04]). The implementation of runtime monitoring as a passive monitor on communication buses in [Kan+15; Kan15] could be used as a reference for the implementation of this thesis' runtime monitoring framework on embedded hardware in production vehicles (cf Chapter 5).
Multiple approaches [Ahl+05; Kan+15; Kan15; KFK14; Rei+11] incorporate temporal logics, e.g., LTL, for the definition and evaluation of temporal requirements, like durations between system events. The runtime monitoring framework in this thesis does not yet consider the temporal requirements but would require a temporal first-order logic similar to[Cim+08; Cim+10] in order to consider the temporal requirements over the objects of the environment.

4.1.2.4. Comprehensive Safety Approaches

Several works integrate runtime monitoring into more comprehensive safety approaches that not only identify faulty system behavior but take corresponding actions in order to mitigate the faulty system behavior.
In [Hör11], Hörwick presents a functional safety concept that combines the monitoring of highly automated ADAS and the transfer of these systems into safe states in the presences of critical and unsafe traffic situations. The safety concept incorporates the monitoring of internal system faults, functional system boundaries, and the human driver. Internal system faults are detected by watchdogs and analyzed for their impact on the system functionality. Functional boundaries of ADAS are derived from their specifications and monitored as constraints on traffic situation parameters. Action plans define the transfer of these systems to a safe state in the presence of possible constraint violations.

Heckmann et al. present in [Hec+11] adaptive software safety cages as a safety-oriented reference architecture for vehicle systems. The safety cage architecture consists of functional safety cages and actuator safety cages. Functional safety cages cover a single control system functions while actuator safety cages focus on safety-related actuators. Safety cages monitor the plausibility of function outputs in the current driving situation and restrict or block these function outputs accordingly. The driving situation is determined based on information such as vehicle speed, driving dynamics, or GPS location. Once a failure is detected, the safety cage invokes an appropriate action in order to transfer the system into a safe state.
The authors of [Ani+18b] propose a similar functional safety architecture. Their dependability cages examine and evaluate the internal behavior of systems and the external behavior of the system environments at runtime based on constraints which have been defined and verified in the system development. Individual dependability cages are compositionally applied to control functions, machine learning functions, subsystems, the complete autonomous vehicle system, and the system environment. Dependability cages record unknown situations for further system improvement and allow to intervene with the system behavior and configuration of the system similar to functional safety cages in [Hec+11].
Wu et al. describe in [Wu+17] the synthesis of safety guards which monitor the input-output behavior of systems and correct any property violation instantaneously. In the presence of property validations, the safety guards process sound and safe system output as a substitution of the violating system output under minimizing the deviation between the two outputs.
The runtime monitoring architecture in this thesis can be used as dependability cages resp. functional safety cages for complex and learning vehicle functions. The overall engineering approach in [Ani+18b] closely matches the holistic engineering approach of this thesis (cf. Section 4.3). Their work has pre-published contents from this thesis.

4.1.3. Synthesis of Verified Vehicle Controllers

Another research direction is the synthesis of a vehicle controller with verified and soundly bounded system behavior. Verification techniques are applied to ensure the safety of the system behavior and exclude any safety risks from the controllers.
Shalev-Shwartz et al. introduce the notion of responsibility sensitive safety (RSS) as global safety model for the behavior of autonomous agents, e.g., autonomous vehicles. RSS formalized the notion of “accident blame”. Autonomous agents play a non-symmetrical role in accidents where one single agent can be “blamed” for the responsibility of the accident. The RSS model includes a formal treatment of “cautious driving” under limited sensing conditions in order to guarantee that no autonomous agent will ever cause an accident of its “blame”. This notion of “cautious driving” is incorporated as constraints for the possible actions into the driving policies of autonomous vehicle systems.
Shalev-Shwartz et al. further define in [SSS17] a formal semantic language that aggregates units, measurements, and action space, into an abstract specification for consideration in the planning of autonomous agents. Such a semantic model is crucial as the computational

complexity of the planning by autonomous vehicle agents does increase exponentially with time and number of agents. The formal semantic language by Shalev-Shwartz is similar to the abstract representation of real-world traffic situations by the runtime monitoring in this thesis.
In [LMT15], Larsen et al. use the tool *Uppaal Stratego* for automatic synthesis of optimal and safe ACC controller. They model the ego vehicle and a leading vehicle in weighted and stochastic timed automata and let them play a game in simulations. Safety properties can be defined and evaluated by querying these games. Therefore, optimal strategies under consideration of safety properties can be derived from these simulations and used for the controller.
Hilscher et al. present in [Hil+11] an approach to prove the safety of multi-lane motorway traffic with lane-change maneuvers by introducing multi-lane spatial logic (MLSL). MLSL allows for purely spatial reasoning about traffic safety on highways by defining an abstract model of multi-lane highway traffic based on spatial properties of local views of cars. This logic allows specifying the space which individual vehicles reserved. Hilscher et al. model corresponding vehicle controllers and show that cars with these controllers occupy and reserve disjoint spaces. In [HLO13], Hilscher et al. extend their work to rural roads.
All three approaches integrate limitations of the system behavior, which are commonly supervised and enforced by runtime monitors, directly into vehicle controllers. These approaches technically ensure that the behavior of systems remains in their safe limits rendering runtime monitoring obsolete. All three approaches manually model the behavior of traffic participants and abstract from real-world behavior. The correctness and safety of the controller are verified in all approaches under the fundamental assumption that the models fully model the real world. However, it is not clear that this assumption holds and the verification results transfer to the real roads because the models have never been validated in the real world.

4.1.4. Related Work from the Field of Avionics

In the avionics domain, autonomous systems have been in operation for some time now. Therefore, verification and validation of autonomous systems have been extensively studied and applied in this domain. The following publications give an overview of similar work in the avionics domain.
Zou et al. combine in [ZAM14] multi-agent simulations and evolutionary search for a safety validation of sense and avoid (SAA) algorithms aboard Unmanned Aerial Vehicles (UAVs). The evolutionary search explores the input space for SAA algorithms and guides the simulation towards challenging situations. The difference to the automotive domain in the autonomous control systems is the increased degree of freedom as an airplane can rise and fall.
The authors of [DG15; Gia+14] present a test environment for the AutoResolver system (cf. [ELC12]). The test environment incorporates a variety of tools, e.g., TestGen and JDart (cf. [HGR13]), to model, simulate, and evaluate scenarios for the loss of separation in air traffic. A loss of separation conflict will be present if two aircraft fly through the same point at the same time. Scenarios for these conflicts are defined by

selecting a point in 3-dimensional space and fly the two conflicting aircrafts backward for a given duration of time. Thus, the individual trajectories for the conflicting aircraft are created. The trajectories are given to the AutoResolver system as test input for finding a resolution of conflict by changing the trajectory for one of the aircraft. The resolution of the AutoResolver system in these tests is monitored for the creation of test cases for secondary conflicts. For the test of a secondary conflict, a point on the resolution trajectory is selected, and a trajectory of another (secondary) aircraft is defined based on this point. In the tests, runtime monitors verify system requirements of the AutoResolver system and log several aspects about the system for later analysis. The monitors are implemented in AspectJ (cf. [Kis02]) in order to prevent any interference with the system's source code.
Brat et al. present in [Bra+15] an extensive case study on a Matlab/Simulink model of a twin-engine aircraft simulation named transport class model (TCM). Requirements from a Boeing 737 (B737) automatic flight systems are used as references for the model. These requirements specified in natural language assumptions about the airplane's control system and its physics on high abstraction level. The Matlab/Simulink model is transformed into a Lustre model (cf. [Cas+87]) while the requirements are decomposed into requirements on the signals of the TCM model and implemented as synchronous observers (cf. [HLR94]) on the TCM model for verification. The proof of some requirements requires the use of compositional verification.
The research in the field of avionics matches the research in the automotive domain. Similar problems are addressed in both domains. These similarities indicate that various domains encounter similar challenges in the development, verification, and validation of autonomous systems. However, neither domains have yet introduced a holistic approach for the development, verification, validation, and operation of autonomous systems which sufficiently addresses the challenges of autonomous systems in unconstrained environments. The engineering approach in this thesis has the potential to be applied in other domains, e.g., avionics or logistics.
The research goal and research questions related to the engineering approach in this thesis are introduced in the following section. The engineering approach itself is present in Section 4.3.

4.2. Emerging Research Questions

The previous section has presented a large variety of related work. This related work addresses partial aspects of shortcomings and problems for the current development process of autonomous vehicle systems in the automotive domain (cf. Section 3.8). Formal proof methods, e.g., model-checking (cf. [Chr08]), do not scale to analyze complete autonomous vehicle systems in their highly dynamic environments, and manual test automation does not scale to the vast number of relevant traffic situation for autonomous vehicle systems (cf. [TMJ12]). Approaches for automatic test generation from real-world data are not sufficiently integrated into the system development and do not sufficiently validate their simulations results in the real world. Most runtime monitoring approach

reason directly on the signals of autonomous vehicle systems without any semantic abstraction and consideration of the system environment — the real world. This low-level monitoring is incapable to thoroughly verify and validate autonomous vehicle systems operating in the real world with a high number of diverse traffic situations.
Essential criteria in the development of autonomous vehicle systems are the correctness and safety — especially for the operation in the real world. In current development practice (cf. Chapter 3), the correctness and safety of autonomous vehicle systems are expressed by corresponding correctness and safety conditions. Autonomous vehicle systems have to satisfy these conditions in order to be judged as correct and safe. This thesis refers to correctness and safety conditions as *qualitative properties* because any of these conditions is either satisfied or violated. Probabilistic estimations of correctness and safety are not part of this work.

Definition 4.1 (Qualitative Properties)**.** *Qualitative properties for autonomous vehicle systems are conditions which these systems have to permanently meet in order to be judged as correct and safe.*

Statements about the correctness and safety of autonomous vehicle systems are only valid in specific contexts, e.g., traffic situations, in which the ground truth has been proven. In the current development practice, qualitative properties are only verified for a finite set of contexts and not for the vast — if not infinite — contexts, which autonomous vehicle systems may encounter during operation in the real world. In open contexts such as the real world, valid statements about the correctness and safety of autonomous vehicle systems require the quantification of their contexts. Therefore, we refer in this work to the scope of valid contexts by safety and correctness arguments as *quantitative scope*. For autonomous vehicle functions operation in the real world, we also denote their context as *environment situations*.

Definition 4.2 (Quantitative Scope)**.** *The quantitative scope is defined as the contexts of systems under which statements about the system's correctness and safety have been established.*

The semantic of *quantitative* in this thesis differs its semantic in other publications. Nenzi et al. define in [Nen+15] a quantitative semantic for the satisfaction of spatiotemporal properties.
Several publications have addressed an improved verification and validation of autonomous vehicle systems (cf. Section 4.1), but none has yet presented any comprehensive engineering approach. None of the published approaches addresses the qualitative and quantitative capabilities for autonomous vehicle systems throughout their complete lifecycle under the consideration of complexity and uncertainty of the real world. The lack of any holistic engineering approach leads to the main research goal of this thesis:

Research Goal: *Enhancement of the current development practice for autonomous vehicle systems into a holistic engineering approach supporting the qualitative and quanti-*

tative supervision and estimation of correctness and safety by the seamless integration of system development and system operation.

The research goal imposes the general necessity for a holistic engineering approach in order to sufficiently estimate the correctness and safety of autonomous vehicle systems operating in the real world. This estimation requires any engineering approaches to determine the contexts (quantitative) in which the correctness and safety of the systems have been evaluated (qualitative). The integration of system development and operation imposes more specific concerns for the autonomous vehicle systems and the engineering approach itself. Further research questions address these concerns.

Research Question 1: *Which are the necessary (technical) foundations for the integration of system development and system operation in order to qualitatively and quantitatively supervise and estimate the safety of autonomous system operating in the real world?*

The development of autonomous vehicle systems will only benefit from the integration of system development and system operation if the commonalities and variations of the autonomous vehicle systems are sufficiently considered. The development approach has to be applicable throughout the complete life-cycle of autonomous vehicle systems. Furthermore, the integration of development and operation for autonomous vehicle systems requires solutions for the definition, evaluation, and estimation of qualitative properties and their quantitative scopes.

Research Question 2: *How to identify and define qualitative properties for each autonomous vehicle system that sufficiently represent its correctness and safety?*

The correctness and safety of autonomous vehicle systems correspond to a set of qualitative properties. A violation of any property indicates an incorrect and unsafe system behavior. Even if engineers assess the correctness and safety of autonomous vehicle systems themselves, they will implicitly assume qualitative properties for their evaluation of the system behavior. For the automatic supervision of qualitative properties, relevant qualitative properties have to be explicitly defined and implemented. The set of properties for each system has to sufficiently represent the different entities and their relations to the system and their environments, e.g., other dynamic objects, that have an impact on the correctness and safety of autonomous vehicle systems. The following research question addresses the automated evaluation for qualitative properties.

Research Question 3: *How to monitor the qualitative properties and their scopes throughout the complete life-cycle of autonomous vehicle systems — from system specification, design, implementation, and verification to operation in the real world?*

For the automatic supervision of the correctness and safety of autonomous vehicle systems, the qualitative properties are commonly implemented as monitors and deployed alongside systems. For autonomous vehicles, these monitors have to automatically evaluate the qualitative properties and their scope based on the available system data. Throughout the complete life-cycle of autonomous vehicle systems, monitors have to

record any violations of qualitative properties and their contexts. Qualitative properties are commonly expressed as system requirements in natural language (cf. [Cle07]) and have to be formalized for the implementation as monitors.

Research Question 4: *How to estimate the residual risks of autonomous vehicle systems during operation in the real world based on the qualitative and quantitative knowledge from their development?*

The correctness and safety of autonomous vehicle systems are unlikely to be verified and validated during their development for the vast — if not infinite — number of environmental situations which these systems may encounter during their operation in the real world (cf Section 3.8.1); residual risks will remain for the operation of autonomous vehicle systems. The runtime monitoring can support the mitigation of risks from autonomous vehicle systems by identifying critical environment situations and unsafe system behavior based on violations of qualitative properties and their scopes.

Research Question 5: *How to record violations of qualitative properties and their scopes during operation in the real world for further analysis and system improvements?*

The improvement of autonomous vehicle systems by any holistic engineering approach requires the recording of any qualitative and quantitative violations. These *qualitative* and *quantitative runtime monitoring* results provide knowledge about insufficient system behavior and its contexts. The knowledge about faulty system behavior in the system evolution supports the identification of corresponding insufficiencies within the systems and their improvement. Possible improvements address elements in the specification, design, implementation, and verification of autonomous vehicle systems.
The following section introduced the holistic engineering approach for the seamless development and operation of autonomous vehicle systems as the solution for the research goal and research questions.

4.3. Seamless Development and Operation of Autonomous Vehicles by Qualitative and Quantitative Runtime Monitoring

Concerning the research goal (cf. research goal), a holistic engineering approach for the seamless development and operation of autonomous vehicle systems is introduced (cf. Fig. 4.1). The holistic engineering approach addresses the correctness and safety of autonomous vehicle systems by bridging the gap between currently applied development, verification and validation techniques for autonomous vehicle systems and their operation in the real world. The approach enables the verification of simulation results and the gathering of additional data for system improvement. Runtime monitoring techniques are used to supervise and the decision making of these systems and record environmental situations.

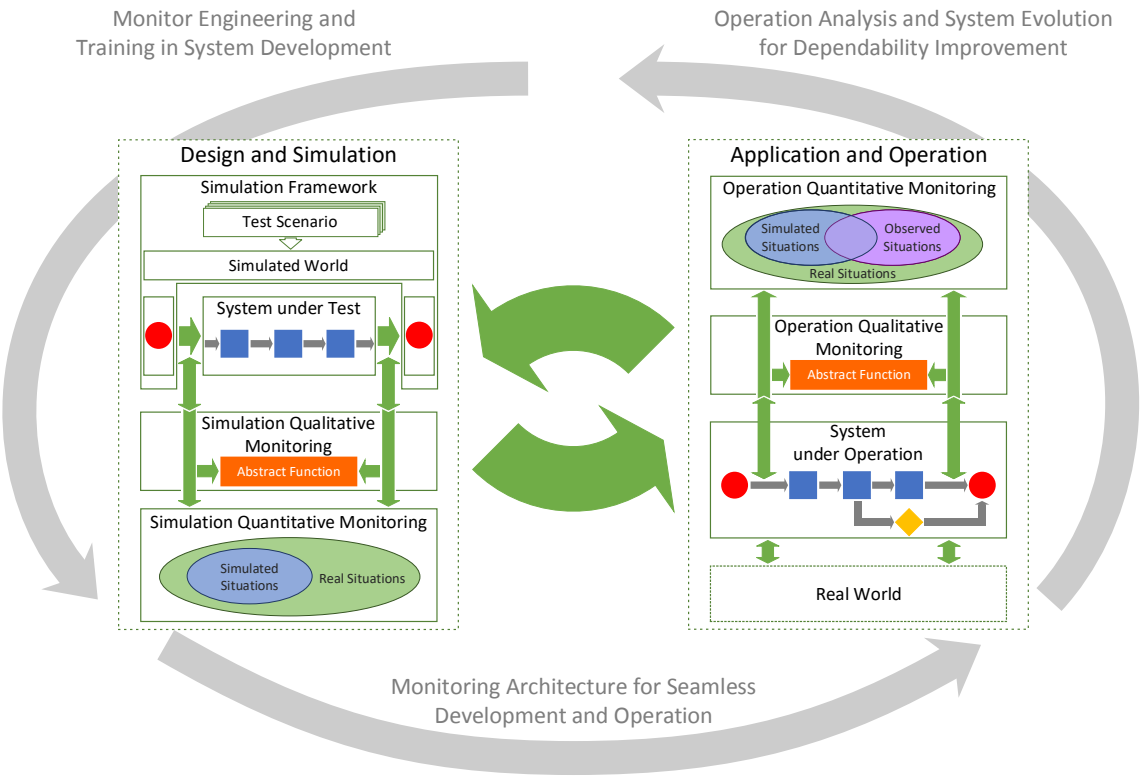


Figure 4.1.: Overview of the engineering approach.

The engineering approach is divided into two parts; a development part (design time) with the design, implementation, and verification of autonomous vehicle systems (cf. left side of Fig. 4.1) and a operation part (runtime) with the application of autonomous vehicle systems in vehicles and their operation in the real world (cf. right side of Fig. 4.1). The approach is not intended to replace the current engineering process in the automotive domain but to extend the current development practice (cf. Chapter 3) by incorporating data from the operation into the system development. Our approach extends the current practice by transferring knowledge about systems and their contexts, e.g., traffic situations, from development to operation and from operation back to the development for further system improvements. Over the complete life cycle of autonomous vehicle systems — from specification, design, implementation, verification to the operation on public roads — the approach enables the continuous improvement of autonomous vehicle systems with particular focus on their correctness and safety. Knowledge from the engineering approach can be used for the estimation of residual risks for the operation of autonomous vehicle systems in the real world.

A key feature of the approach is the usage of runtime monitors for the seamless integration of development and operation. Runtime monitors observe a system (part) via defined interfaces and evaluate predefined conditions and invariants about the system behavior based on data from these interfaces (cf. Definition 2.33).

Information about the autonomous vehicle systems and their contexts are gathered in the specification, safety analysis, system design, implementation, and verification. The runtime monitors use this information for the supervision of autonomous vehicle systems during operation in the real world. Novel information about the systems and their contexts from the runtime monitoring during operation in the real world are transferred back to the development for further improvement of specification, design, implementation, and verification of the autonomous vehicle systems. This process enables the iterative improvement of autonomous vehicle systems, their correctness and their safety over multiple system generations.

The engineering approach focuses on the externally observable behavior of autonomous vehicle systems. This external observable behavior is implemented as software components and emerges from the execution of these software components on the systems' hardware components, e.g., ECUs. Hardware related characteristics, e.g., utilization, timing behavior, or performance of bus communication systems and processors, are not explicitly targeted by the approach. Nevertheless, these characteristics have an impact on the correct and safe system behavior and may be incidentally detected by the runtime monitoring. For example, delayed responses within the processing chain of autonomous vehicle systems may lead to unsafe system behavior. Autonomous vehicle systems have to process their response in time before the real world has evolved into another situation in which the system response is inappropriate.

The holistic engineering approach can be segmented into three subprocesses; the definition of the functional architecture for the runtime monitoring, the definition, implementation, and training of the runtime monitors in the system development, and the analysis of the system operation for system evolution (cf. Fig. 4.1). Each subprocess is briefly described in the following sections Sections 4.3.1 to 4.3.3 and in more detail in the chapters Chapters 5 to 7.

4.3.1. Monitoring Architecture for Seamless Development and Operation

Essential for the seamless integration of system development and system operation is a consistent architecture of the autonomous vehicle system and the runtime monitoring framework throughout the complete life-cycle — from the system verified in simulations to the validation during operation in the real world. The vehicle system has to provide the runtime monitoring framework with consistent interfaces for the access of necessary system data. These interfaces also define the specific part of the system which is supervised by the runtime monitoring.

The runtime monitoring abstracts from the data of the autonomous vehicle systems (cf. [LF96]). The data accessed by the runtime monitors via the consistent interfaces of the autonomous vehicle system is transformed into an abstract representation. During development and operation of autonomous vehicle systems, the behavior of the supervised *system part* is qualitatively and quantitatively monitored based on the abstract representation:

Qualitative Monitoring: An *abstract function* supervises the behavior of systems by implementing and evaluating the systems' qualitative properties (cf. *Simulation Qualitative Monitoring* and *Operation Qualitative Monitoring* in Fig. 4.1).

Quantitative Monitoring : The runtime monitoring framework determines if encountered instances of the abstract representation have already been considered and verified in the system development (cf. *Simulation Situation Monitoring* and *Operation Situation Monitoring* in Fig. 4.1).

Definition 4.3 (Abstract Representation)**.** *The* abstract representation *represents the abstract data model used by the runtime monitoring framework for the abstraction of the operational data of autonomous vehicle systems. Elements of the abstract representation are derived for each autonomous vehicle system from its specification, requirements, and safety criteria.*

The architecture of the runtime monitoring framework and its integration with the autonomous vehicle systems are described in more detail in Chapter 5.

4.3.2. Monitor Engineering and Training in System Development

While the architecture defines the components of the runtime monitoring framework, the abstract representation and properties for correct and safe system behavior are subject to each particular system (part) under monitoring. The abstract representation and qualitative properties are individually defined for each system (part). Besides other specialized (safety) documents (cf. [Int09c]), *system specifications* commonly aggregate requirements which define the correct and safe behavior of the system.
Requirements are commonly described in natural language. Types and relations used by the requirements to describe correct and safe system behavior have to be formalized for their usage in the runtime monitoring. The types represent the abstract representation while the relations constitute the properties for the runtime monitoring. The logic forms the foundation for the implementation of abstraction, qualitative motoring, and *quantitative runtime monitoring* in the runtime monitoring framework (cf. Chapter 6).
All requirements are required to be defined in a predefined sentence structure. This sentence structure allows defining the relations between data types of autonomous vehicle systems and elements of the logic. These relations are implemented by the runtime monitoring framework for the automated transformation of system data into the abstract representations. The defined sentence structure also helps to mitigate the ambiguity of natural language in the definition and selection of requirements.
For the system verification in the system development, the autonomous vehicle system and the runtime monitoring framework are integrated into simulations (cf. left side of Fig. 4.1). In these simulations, autonomous vehicle systems are verified for a set of defined test cases (cf. Section 3.6). The runtime monitoring framework is deployed in these simulations alongside the autonomous vehicle systems (cf. Fig. 4.1) and accesses at each processing cycle the real data of a monitored system part via defined interfaces. The system data is

transformed into an instance of the abstract representation. In the course of this work, we denote an instance of abstract representation for the state of autonomous vehicle systems and their environments as *abstract situation*. The qualitative properties are evaluated on each *abstract situation* by the *abstract function* (cf. simulation qualitative monitoring in Fig. 4.1) in order to evaluate the correctness and safety of the system part under monitoring. The *quantitative runtime monitoring* records all encountered *abstract situations* in the test cases. This set of simulated situations represents only a subset of situations which autonomous vehicle systems may encounter in the real world (cf. Fig. 4.1).

Definition 4.4 (Abstract Situation / Abstract Representation)**.** Abstract situation *defines an instance of the abstract representation for the state of the system and its environment in the runtime monitoring framework.*

The definition of the runtime monitoring framework from the system specification and its application in the system simulations of the system verification are described in more detail in Chapter 6.

4.3.3. Operation Analysis and System Evolution for Dependability Improvement

Besides the verification of the autonomous vehicle system in simulations, the runtime monitoring is also used during the validation of autonomous vehicle systems in the real world (cf. right side of Fig. 4.1). Similar to the system verification in simulations, the runtime monitoring framework access the system data via the identical data interfaces as in the simulations and processes an abstract representation for each processing cycle of the system. The *abstract function* evaluates the correctness of the system part under monitoring based on the identical properties as they have been used in the simulations (cf. operation qualitative monitoring in Fig. 4.1). The *qualitative runtime monitoring* during operation enables the identification and mitigation of faulty system behavior during operation — even in situations which have not been considered and verified in the development of the system. The *quantitative runtime monitoring* uses the set of simulated situations as the reference set for the monitoring of situations during operation in the real world. During operation, the reference set is compared to the encountered *abstract situations* and any *abstract situations* which have not yet been considered in the system development are recorded (cf. operation quantitative monitoring in Fig. 4.1). Neither the set of simulated situations nor the set of encountered situations during operation will fully include all situations which are possible in the real world (cf. Section 3.8.1).
The knowledge about unconsidered *abstract situations* and faulty system behavior can be used for the improvements of the specification, design, implementation, and verification for autonomous vehicle systems in further developments. An extensive V&V represents a key factor for the successful development of correct and safe autonomous vehicle systems. Our engineering approach supports the testing of autonomous vehicle systems

in simulations by enabling the definition of new *test scenario* and *test cases* from the recorded *abstract situations* during operation in the real world. The new *test scenarios* and *test case* are envisaged to increase the *test coverage* of the autonomous vehicle systems by verifying the behavior of these systems in the unknown traffic situations.
Test scenarios describe timed sequences of scenery changes and maneuvers by dynamic objects in the environment of the autonomous vehicle system. These sequences are identified in the recorded traces of *abstract situations* from the runtime monitoring of the autonomous vehicle system during operation in the real world.

Definition 4.5 (Test Scenario)**.** *A test scenario defines a timed sequence of scenery changes and maneuvers by dynamic objects in the vicinity of the system under test (SUT) for simulation-based tests.*

Test cases are derived from test scenarios by parametrization. Parameters of a test scenario still correspond to the *abstract situations* and do not correlate to the required concrete parameters of simulation frameworkssimulation framework. For example, *abstract situations* may only reason about the relative velocity between two vehicles whereas the simulation framework requires the actual velocity of each vehicle for its simulations (cf. Chapter 8). The *parametrization* estimates missing concrete parameters for the simulations from the available parameters of *abstract situations* in the test scenarios. Novel *abstract situations* in the simulations of these new test cases extend the reference set for the runtime monitoring of autonomous vehicle system during operation in the real world.

Definition 4.6 (Test Case)**.** *A test case represents the parametrization of a test scenario in which all necessary parameters for the simulation have been defined.*

The analysis of the operation of autonomous vehicle systems in the real world with by the runtime monitoring framework and the usage of runtime monitoring results for the evolution of autonomous vehicle systems are described in more detail in Chapter 7.

5. Monitoring Architecture

Runtime monitors represent the crucial part of the holistic engineering approach for autonomous vehicle systems (cf. Section 4.3). The runtime monitoring framework enables to qualitatively and quantitatively evaluate the behavior of autonomous vehicle systems and to record corresponding data for the seamless transfer between system development and system operation. Nevertheless, the seamless transfer of operational data requires a consistent architecture of the autonomous vehicle systems and the runtime monitoring framework throughout the complete system life-cycle. The architecture of the corresponding runtime monitoring framework consists of five layers (cf. Fig. 5.1).The two bottom layers address the basic structure of autonomous vehicle systems and their integration into simulation frameworks for their verification (cf. Section 3.6). The upper three layers represent the runtime monitoring framework.

Many runtime monitoring techniques exists with their individual benefits and drawbacks (cf. Section 2.3). The runtime monitoring framework in this work represents an external runtime monitoring, that does not require any instrumentation of the systems' source codes. The framework non-intrusively monitors the behavior of the autonomous vehicle systems solely based on the operational data that is available by defined interfaces from these systems. The autonomous vehicle systems present themselves as black-boxes to the runtime monitoring framework — no knowledge about the internal structure of these systems nor any internal data and control flows is available for the runtime monitoring framework. Autonomous vehicle systems have to provide the interfaces consistently throughout the complete life-cycle (cf. Fig. 2.2) in order to enable the runtime monitoring framework to access the systems' runtime data persistently. The correctness and safety of autonomous vehicle systems are qualitatively and quantitatively evaluated by the runtime monitoring framework based on the abstraction of the provided runtime data (cf. Fig. 5.1). In the following sections, we will describe each layer of the runtime monitoring architecture in more detail. We commence with the description of the *system layer*.

The runtime monitoring is executed alongside the autonomous vehicle systems (cf. Fig. 5.1). Runtime monitoring and systems may share the same hardware components, like processors, memory, and bus communication systems. For the validity of the runtime monitoring, the runtime monitors must not interfere with the operation of the autonomous vehicle systems. Otherwise, the behavior of the monitored autonomous vehicle systems may diverge from the behavior these systems will exhibit during operation without the runtime monitoring. In such a case, results from the runtime monitoring are not applicable to the general operation of these systems without the runtime monitoring. Following [GP10], four properties are defined for the runtime monitoring framework:

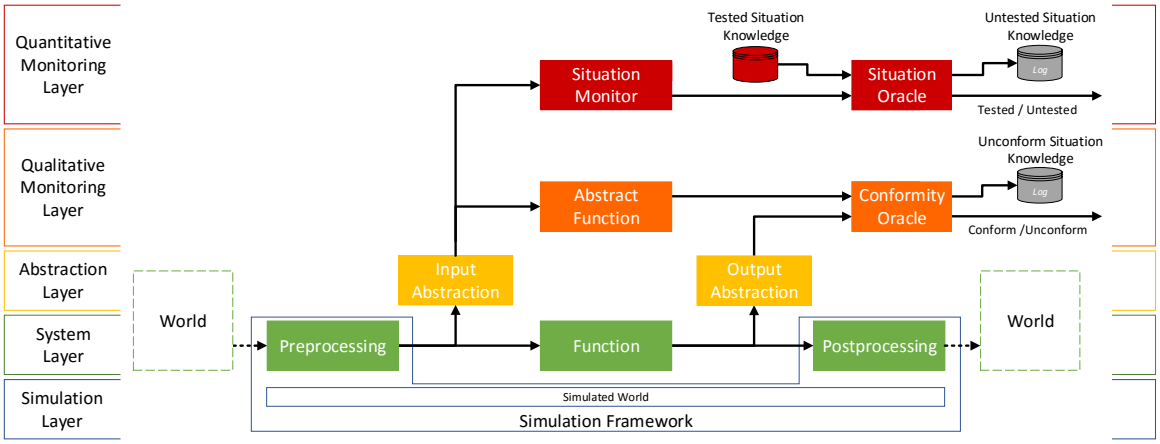


Figure 5.1.: Monitoring architecture for simulations at design time.

Dependability The dependability of the autonomous vehicle system is equal or greater compared to the dependability of the systems alone. The runtime monitoring must not introduce any faults that will diminish the overall dependability.

Functionality The runtime monitoring must not introduce, change, or remove any functionality of the monitored autonomous vehicle systems.

Schedulability The runtime monitoring must not lead to violations of real-time guarantees for the monitored systems.

Certifiability The runtime monitoring does not require instrumentation or other modification for the source code of the autonomous vehicle systems.

The only context in which the runtime monitoring is allowed to violate these properties is a specification violation by the monitored autonomous vehicle systems. In case the runtime monitoring detects incorrect and unsafe system behavior, safety measures have to be initiated in order to reduce the emerging safety risks by the faulty system. As the original behavior of the system is already compromised, the transfer of the corrupted systems into a *safe state* with acceptable risk has the highest priority (cf. Definition 2.8) Safety measures are considered in this thesis as part of the runtime monitoring, but their selection and execution are not explicitly addressed. The selection and execution of appropriated safety measures in critical situations are too complicated in order to sufficiently address it in this work. The reader is referred to other work in this field, i.e., [Hör11; RM15].

5.1. System Layer

The *system layer* represents the basic structure of the autonomous vehicle systems (cf. Section 2.1.3). As control systems, autonomous vehicle systems have to continuously

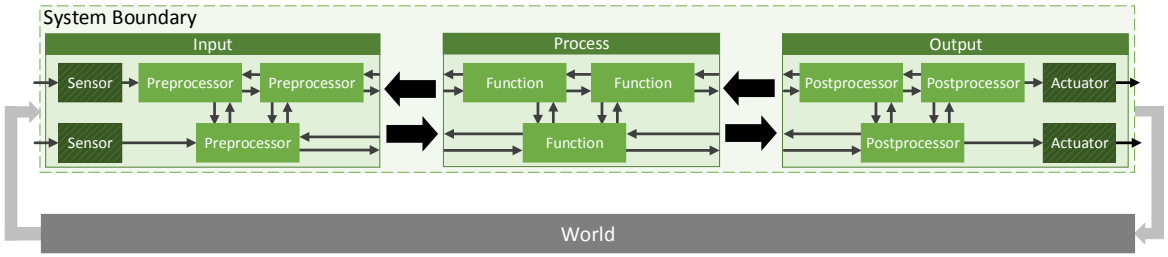


Figure 5.2.: The segmentation of autonomous systems following the IPO model.

adapt the behavior of in reaction to changes by the ego vehicle and the world, e.g., other dynamic vehicles (cf. Fig. 5.2). Feedback loops allow these systems to consider changes of other vehicles in the environment in their processing (cf. Section 2.1.5). Autonomous vehicle systems manipulate the position and pose of vehicles in the world based on information about the environment from the vehicle sensors. Changes of the autonomous vehicle systems for the position and pose of the vehicles in the environment are executed by the vehicles' actuators, e.g., brakes, steering, and engine. Sensors and actuators represent the interfaces between the physical world and the digital systems. As shown in Fig. 5.2, autonomous vehicle systems can be split into three segments — the *input*, *process*, and *output* segment. This segmentation originates from the input-processing-output (IPO) model (cf. [Goe10]) correlates to the architecture of control systems (cf. Fig. 2.5).

The *input* segment addresses the perception of the own vehicle and the real world by proprioceptive and exteroceptive sensors. The perception includes the preprocessing of sensor data into a comprehensive representation which acts as the input to the process segment. For the lane change assistant (cf. Section 3.1), the *input* segment correlates to the environment perception (cf. Section 3.3.1.1).

The *process* segment resembles the main control algorithms for the *behavior planning* and *decision making* of the autonomous vehicle systems. Based on the input from the *input* segment, changes for the behavior of the vehicle are processed and forwarded to the output segment. For the lane change assistant, the *process* segment correlates to the *situation assessment* (cf. Section 3.3.1.2) and *behavior planning* (cf. Section 3.3.1.3).

The *output* segment addresses the execution of planned behavior by vehicle actuators. The intended behavior from the *process* segment is preprocessed into inputs for the different actuators, e.g., brakes, gearbox. The execution of actuators leads to a change of the vehicle state in detail and the world state in general — including states of surrounding dynamic objects. These state changes require the autonomous vehicle systems to revise their assumptions about the world state reiteratively and to adapt the vehicles' behaviors. For the lane change assistant, the *output* segment correlates to the modules of the vehicle *stabilization* and hardware actuators (cf. the postprocessing of target points in Section 3.3.1.3).

Though the segmentation of autonomous vehicle systems may vary between different system instances (cf. Fig. 5.2), the three segments — *input*, *process*, and *output* — will exists for any autonomous vehicle system. Sensors — *input* — and actuators — *output* —

are mandatory for the interaction of the systems with the real world. For the vehicle control, a function — *process* — has to process targets for the vehicle actuators based on the inputs from the vehicle sensors.

Autonomous vehicle systems consist of multiple software and hardware components (cf. Fig. 2.7). Each component implements a portion of the system's functionalities. Components integrated with each other into a component *hierarchy* for the autonomous vehicle systems (cf. Section 3.3). Hardware parts, e.g., ECUs, are assembled from multiple other hardware parts, e.g., CPU and memory (cf. Section 3.3.2). Software components are either *atomic* or *composites* of other interacting software components (cf. Section 3.3.1). Each component of autonomous vehicle systems can be assign to one of the three segments — *input*, *process*, and *output*.

The following section describes the *simulation layer* and its substitutions in the verification of autonomous vehicle systems.

5.2. Simulation Layer

The *simulation layer* addresses the integration of autonomous vehicle systems into simulations. Software-based system simulations are used in the automotive domain for the verification of autonomous vehicles systems. Simulation frameworks imitate the environments of these systems — the *world* — including their sceneries and dynamic objects (cf. Section 2.2).

The integration into the simulation segments the autonomous vehicle systems into three parts (cf. Fig. 5.3); the part of the original system whose functionality is verified in the simulation and two parts of the system that are simulated by the simulation framework. The segmentation depends on the system functionality which is verified. The verified system part is also denoted as system under test (SUT) (cf. Definition 2.16). All system parts which are simulated by the simulation framework, e.g., sensors and actuators, have to be verified and validated by other V&V methods.

The segmentation of the autonomous vehicle system in the simulation defines the input and output interfaces for the interaction between system and simulation framework. For tests, the SUT is stimulated by input data from the current state of the virtual world. The output of the SUT is manually and automatically evaluated for a verdict about the system's correctness (cf Section 2.2). In *closed-loop* simulations, the state of the virtual world is also adapted to the output of the SUT. The interfaces between the system part and simulation framework define the maximal extension of the system segment that can be monitored by the runtime monitoring (cf. Fig. 5.3).

The integration of autonomous vehicle systems and the virtual world requires simulation frameworks to substitute and simulate systems components by models (cf. Fig. 5.3). The extent of simulated components depends on the segmentation of the autonomous vehicle system in the simulation. Simulations impose different requirements for the availability of real software and hardware components (cf. Section 2.2). Models substitute sensor and actuator components in software-based simulations because they rely on hardware

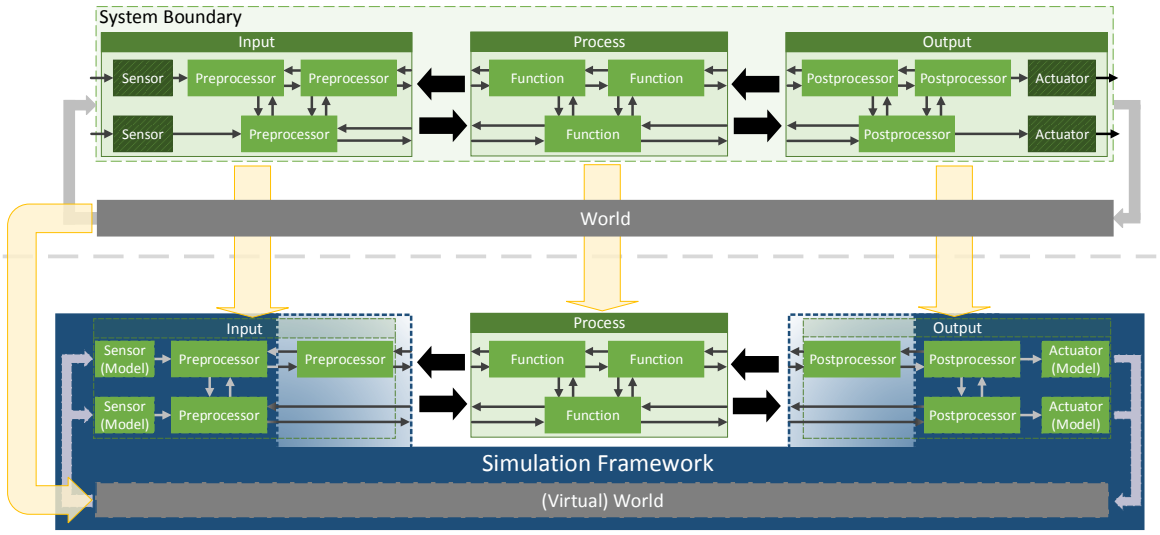


Figure 5.3.: Integration of autonomous vehicle system and simulation framework.

and real-world objects. Hardware components are commonly verified independently in HIL tests (cf. Section 2.2.2.4).

Sensors, e.g., RADAR and LIDAR sensors, require the reflection of real-world objects for their physical measurements. However, objects in software-based system simulations are mostly virtual (cf. Section 2.2). The real propagation of radio waves and light beams is impossible to be included in these simulations. These sensors have to be simulated by corresponding sensor models. Sensor models with different degrees of accuracy can be included in simulations (cf. Section 2.2.1.2). Perfect sensor models use the parameters of objects in the virtual world as outputs of the sensor components and direct input for the SUT. Intermediate sensor models introduce measurement noise for the parameters of the virtual objects in the input for the SUT. Complex models may use techniques, e.g., ray-tracing (cf. [GGD12]), to model the propagation of radio waves and light beams in the virtual world.

The integration of actuators, e.g., brakes and engines, into software-based simulations, require extensive hardware setups. In *closed-loop* simulations, the physical behavior of actuators has to be interwoven with the virtual behavior of the vehicles in these simulations. Additional hardware has to physically represent the virtual world of the simulation frameworks to these actuators. For example, virtual road parameters, e.g., slope, would have to be applied as resistance on the crankshaft of the real engine for a realistic engine behavior. Virtual parameters of objects in the simulations have to be incorporated into the physical behavior of real hardware components. The costs for such complex hardware setups would neglect the benefit of software-based system simulations in comparison to real-world tests. Therefore, models substitute the real sensors, actuators, vehicle dynamics, and road characteristics in software-based simulations.

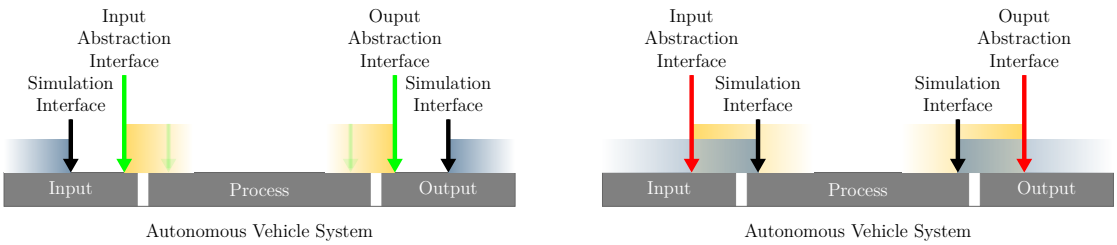
Inaccurate simulations restrict the validity of their results for the operation of autonomous vehicle systems in the real world. The system behavior in the real world has to match

the behavior of autonomous vehicle systems in simulations. The overall accuracies of simulations predominantly depend on the accuracy of the simulation models to represent the physical properties of real-world objects and real-world processes in the simulations. The output of simulation models must not significantly diverge from the output of the components during operation in the real world (cf. [HK16; Ste+15]). The accuracy of simulation models has to be verified by comparing the outputs of the models with their corresponding real components for the same real-world inputs.
The impact of insufficient simulation accuracy for the engineering approach is limited. The approach expects engineers to evaluate the results of the simulations before the knowledge from these simulations is used in the runtime monitoring. In case, simulations have not been accurate for the real world, or the behavior of autonomous vehicle systems has not been correct and safe, data from these simulations must not be used as knowledge for the runtime monitoring during operation. The *qualitative runtime monitoring* can be used as test oracle in simulations without any concurrent evaluation engineers as long as the monitors have been successfully verified.
The following sections describe the layers of the runtime monitoring framework in detail. We commence with the *abstraction layer* as the interface of the runtime monitoring to the autonomous vehicle systems.

5.3. Abstraction Layer

The *qualitative* and *quantitative monitoring* of autonomous vehicle systems require the consistent access of system data throughout system development and system operation. The *abstraction layer* (cf. Fig. 5.1) depicts the interface between the autonomous vehicle systems and the runtime monitoring framework. The layer consists of two components — the *input abstraction* and *output abstraction*. These components access the runtime data of the autonomous vehicle systems via defined interfaces and transform the runtime data into the abstract representations of the runtime monitoring. Types and values of the abstract representation are defined from the specification, requirements, and safety criteria of the autonomous vehicle systems (cf. Chapter 6). The correctness and safety of autonomous vehicle systems is evaluated by the *qualitative* and *quantitative* monitoring layers based on this abstract representation (cf. Fig. 5.1).
The part of the autonomous vehicle systems which is supervised by the runtime monitoring framework is denoted as *function* (cf. Fig. 5.1) or as *system (part) under monitoring*. Engineers define the *function* during the system development in relation to a specific system functionality. The *function* need not correlate with the *process* of autonomous vehicle systems (cf. Section 5.1) and, therefore, may vary between different autonomous vehicle systems. For one system, the *function* may incorporate components of the system's *input* and *output* parts while the runtime monitoring of another system solely focuses on a subset of components within this *process* par as the *function*.
The runtime monitoring framework depicts the *function* and its components as a *black-box*. The *function* is solely monitored based on its external observable behavior without any consideration of its internal structure. The hierarchical architectures of autonomous

vehicle systems have no impact on the runtime monitoring as long as the required input and output interfaces of the respective *function* are provided.
The processing of autonomous vehicle systems before the function is depicted as *preprocessing* and mostly consists of the sensor perception and the processing of the physical sensor data. The system components in the system's processing chain following the *function* is encompassed by the *postprocessing*. The *postprocessing* addresses the execution of the function's output by actuators in the real world. The correctness and safety of any component within the preprocessing and postprocessing are not addressed by the runtime monitoring (cf. Fig. 5.1). Components of the preprocessing and postprocessing, e.g., sensors and actuators, have to be verified and validated by other methods.
Qualitative and quantitative statements by the runtime monitoring are only valid for the *function*. For the transfer of monitoring results between system development and operation, the *function* has to remain consistent. The same set of system components has to be encapsulated by the *function* throughout system development and operation. Otherwise, any statements about the systems' correctness and safety from simulations are not applicable during the operation of autonomous vehicle systems in the real world. Situations recorded in simulations cannot be used as reference knowledge in the runtime monitoring during operation because the behavior of the system version during operation has not been verified. Without the runtime monitoring, it will be impossible to determine if two different versions of an autonomous vehicle system are behaviorally equivalent.
The *abstraction layer* impacts the implementation of the runtime monitoring properties — dependability, functionality, schedulability, and certifiability. The runtime monitoring framework does not affect the functionality of autonomous vehicle systems because the data access via interfaces separates the concerns of autonomous vehicle systems and runtime monitoring framework. The system interfaces for the runtime monitoring are the only connections between the systems and the runtime monitoring framework. No functionality is added, changed, or removed to/from the autonomous vehicle systems by the application of the runtime monitoring as long as the provision of data for the runtime monitoring within autonomous vehicle systems has no impact on their functionality. The non-intrusive data access via the interfaces also addresses the certifiability. In case the source code of autonomous vehicle systems has been certified, the application of the runtime monitoring must not require a full re-evaluation of the compound system of the autonomous vehicle system and runtime monitoring framework (cf. [GP10]).
The overall dependability of autonomous vehicle systems requires their functional integrity (functionality) and their integrity of the timing behavior (schedulability). The deployment of the runtime monitoring framework along autonomous vehicle systems has to ensure the schedulability. The processing and communication of the runtime monitoring framework have to be integrated into the scheduling of the systems' processors and bus communication systems without any violations of systems' real-time requirements. One solution would be the deployment of the runtime monitoring framework and the autonomous vehicle systems on separate processors or separate ECUs with dedicated communication between them. A solution for shared processors and bus communication systems would be to restrict the computation by the runtime monitoring framework to idle segments in the scheduling of the autonomous vehicle systems.



(a) Correct configuration of interfaces.

(b) Incorrect configuration of interfaces.

Figure 5.4.: Configurations of system interfaces.

The interfaces of autonomous vehicle systems for the runtime monitoring are discussed in more detail in the following section.

5.3.1. System Interfaces

The *function* has to be encompassed by consistent interfaces. As shown in Fig. 5.4a, input and output interfaces for the runtime monitoring in an autonomous vehicle system have to reside within the interfaces of the system to the simulation framework. The extent of autonomous vehicle systems, which is present in simulations in original form, limits the maximal possible size for the *function*.

In case runtime monitoring interfaces reside outside the valid system extend, additional interfaces between the runtime monitoring framework and the simulation frameworks have to be introduced. These interfaces have to provide the runtime monitoring framework with the identical data as corresponding system components would provide during operation (cf. Fig. 5.4b). Models in the simulations substitute the original components of the system outside the monitored system part. These simulation models may miss the necessary granularity by their processing in order to provide the runtime monitoring framework with the required data quality and granularity. For example, the relation between vehicle actuators, vehicle dynamics, and road conditions for the vehicle behavior and their impact on the virtual world may all be simulated by one single model.

In the automotive domain, the reduction of costs is an essential objective in the development and production of vehicles and their systems (cf. [SZ13]). Hardware costs represent a large quantity of the overall costs of vehicles. For the usage of less expensive hardware components, e.g., ECUs or wiring harness, the performance of vehicle systems — timing behavior, memory, and energy consumption — is highly optimized. These optimized vehicle systems offer little to no additional resources for the generation of additional monitoring data. Unless the intrusive data generation has been explicitly considered in the system development, any intrusive alteration of autonomous vehicle systems for the generation of runtime monitoring data, e.g., source code instrumentation (cf. Section 2.3.1), is not applicable. The memory and processor consumption of code instrumentation would interfere with the functionality of the autonomous vehicle systems.

The runtime monitoring depicts the *function* as black-box and only evaluates its external observable behavior. The *input abstraction* and *output abstraction* access the system data via defined interfaces of the autonomous vehicle systems. No code-instrumentation is required for the access to the system data. The interfaces for the input and output data have to be defined in the development of autonomous vehicle systems explicitly for the runtime monitoring or already exist within the systems' architectures. Technically, these interfaces can be implemented by e.g., shared memory or access to bus communication and may incorporate frameworks, e.g., ADTF (cf. Section 2.3.1).

The data abstraction is the central functionality of the runtime monitoring and is described in more detail in the following sections.

5.3.2. Data Abstraction

The data abstraction aims to enhance the runtime monitoring and its evaluation of qualitative invariants by reducing the complexity of data. For each autonomous vehicle system, the abstract representation is defined from the system's specifications, requirements, and safety criteria — their entities, objects, measures, and units (cf. Chapter 6). The *input abstraction* and *output abstraction* reduce the high-dimensional data space of the *function* to the abstraction level used in the system requirements and safety criteria. The runtime monitoring framework evaluates the correctness and safety of autonomous vehicle systems on the same abstraction level that engineers have been using for reasoning about the correct and safe behavior of autonomous vehicle systems in the *requirements analysis* and *safety analysis* (cf. Sections 3.2 and 3.4).

The evaluation of qualitative invariants by the runtime monitoring requires the accessible data of autonomous vehicle systems to sufficiently represent the current *system state* (cf. [KFK14]). The architecture of the runtime monitoring framework addresses this issue by its abstraction of system data (cf. Fig. 5.1). Parameters of the abstract representation do not have to identically exist in the accessible system data. Each abstract parameter can be calculated in the abstraction from available interface data of the *function*. These calculations might only be able to model real processing of these hidden state variables in autonomous vehicle systems (cf. Section 3.3.1.2.3).

The abstraction of the runtime monitoring must be deterministic and sound. It has to preserve the distinction between safe and unsafe system behavior. Only safety unrelated information may be discarded from the input space of the *function*. The abstraction has to deterministically and soundly abstract all safety-related elements in the system data. Each element must always be abstracted to the identical abstract parameters in the abstract representation of the runtime monitoring framework. Entities in the system data, which relate to different system behavior — safe and unsafe system behavior —, must not be identically classified into the same abstract parameter.

The *abstraction layer* consists of two components — the *input abstraction* and *output abstraction*. Each component is described in more detail in the following sections.

5.3.2.1. Input Abstraction

The component *input abstraction* of the *abstraction layer* encapsulates the transformation between the input data of the *function* and the abstract representation used for *qualitative* and *quantitative runtime monitoring* (cf. Fig. 5.1). As described in detail later in Chapter 6, data types, and values of the *abstract representation* are derived from the system requirements and safety criteria.

An abstraction formula is defined for each data type of the abstract representation. The abstraction formula defines the calculation of an abstract data type from the input data of the *function*. In every processing cycle, all abstraction formulas of the *input abstraction* are processed in order to update the values for the abstract representation from the current data of the *function's* input interface — the *state*state of the autonomous vehicle system and its environment. The output of the *input abstraction* — an instance of the abstract representation — is used by the *abstract function* of the *qualitative monitoring layer* to process the set of safe system action. The current instance of the abstract representation is compared by the situation monitoring with its knowledge of known and verified *abstract situations*.

The abstraction of system data for the output of the *function* is described in the following sections.

5.3.2.2. Output Abstraction

Similar to the *input abstraction*, the *output abstraction* process an abstract representation of the *function's* output (cf. Fig. 5.1). The abstract representation for the *output abstraction* is derived from the same requirements and safety criteria as used for the *input abstraction*. The abstract representation of the *function's* output is also denoted as *abstract action* or *abstract behavior*. For each processing cycle, the intended system behavior as output of the *function* is processed into the *abstract representation*. The intended system behavior has to be represented on the same abstraction level as the abstract representation of the *input abstraction*. For the *qualitative runtime monitoring* (cf. Section 5.4), the *abstract action* of the *function* is compared by the *conformity oracle* to the output of the *abstract function* — the set of safe *abstract behavior* by the autonomous vehicle system (cf. Fig. 5.1).

Definition 5.1 (Abstract Action / Abstract Behavior)**.** Abstract action *or* abstract behavior *defines an instance of the abstract representation in the runtime monitoring framework for the* output *of the* function *resp. the* behavior *of the autonomous vehicle system.*

The structure of the *qualitative* and *quantitative runtime monitoring* layers and their usages of the abstract representations are described in the following sections.

5.4. Qualitative Monitoring Layer

The *qualitative runtime monitoring* evaluates the correctness and safety of the *function* on an abstract level. It consists of the *abstract function* and the *conformity oracle* (cf. Fig. 5.1). The *abstract function* processes a set of correct and safe actions for the *function* in the current *abstract situation*. The *conformity oracle* compares the abstracted output from the *function* with the set of correct and safe abstract actions from the *abstract function*.

The components of the *qualitative monitoring layer* are identically used in simulations in simulations of the system verification and during operation of autonomous vehicle systems in the real world. In simulations, the *abstract function* and *conformity oracle* are used as test oracles. They automatically evaluate the behavior of the *functions* in these simulations (cf. Section 6.2.3.4.2). In the early stages of the application in the simulations, results of *qualitative runtime monitoring* should be validated by engineers in order to ensure the correct implementation of the abstraction, *abstract function*, and *conformity oracle*. During operation in the real world, the *qualitative runtime monitoring* acts as a safety monitor (cf. Section 7.1.1). In case the *qualitative runtime monitoring* detects any unsafe behavior of the *function*, safety measures can be initiated and the *system state* of system and environment can be logged for later analysis.

The components of the *qualitative monitoring layer* — *abstract function* and *conformity oracle* — are described in more detail in the following sections.

5.4.1. Abstract Function

The *abstract function* represents a second instance of the system's *function*, which reasons about the *function's* behavior on the abstraction level (cf. Fig. 5.1). Opposed to the *function* of the autonomous vehicle system, the *abstract function* does not process one single action for the system but determines a set of correct and safe actions based on the current *abstract situation*. The *abstract situation* is processed by the *input abstraction* (cf. Section 5.3.2.1).

The *abstract function* is defined based on the qualitative invariants contained in the specification, requirements, and safety criteria of autonomous vehicle systems (cf. Chapter 6). While the entities of the requirements and safety criteria are used to define the data types and their values for the *input abstraction* and *output abstraction*, the *qualitative invariants* over these data types are implemented by the *abstract function*. The evaluation of the qualitative invariants by the *abstract function* leads to the exclusion of incorrect and unsafe actions for the *function*. The output of the *abstract function* is the set of correct and safe abstract actions for the *function*. The output of the *abstract function* is used by the *conformity oracle* to evaluate the actual output action of the *function*. The *conformity oracle* is described in the following section.

5.4.2. Conformity Oracle

The *conformity oracle* evaluates the correctness and safety of the *function* by comparing the actual output of the *function* with the output of the *abstract function* (cf. Fig. 5.1). The oracle will evaluate the behavior of the *function* as correct and safe, if the abstracted output action from the *function* resides within the set of safe and correct abstract actions processed by the *abstract function*. The comparison of the *conformity oracle* includes the check of *abstract actions* from the *abstract function* and *output abstraction* for equivalence. Some use cases may require to consider probability distributions in the comparison by the *conformity oracle* instead of object equivalence. Before this comparison, the actual output action of the *function* is preprocessed by the *output abstraction*.

In case, the behavior of the *function* has been judged as unsafe by the *conformity oracle*, the *abstract action* and corresponding *abstract situations* are logged for further analysis (cf. Fig. 5.1). The verdict of the *conformity oracle* is further used during operation for the initiation of safety measures in order to reduce the emerging risks of unsafe *function* behavior. The selection and execution of appropriate safety measures for a critical situation can become very complex. For this reason, selection and execution of safety measures are beyond the scope of this thesis.

In addition to the *qualitative monitoring* of the correctness and safety, the runtime monitoring framework also monitors the encountered *abstract situations* in order to quantify the scope of its *qualitative monitoring statements*. The structure of the *quantitative runtime monitoring* are described in the following section.

5.5. Quantitative Monitoring Layer

The *quantitative monitoring layer* assesses the scope of qualitative statements about autonomous vehicle systems. Qualitative statements can either be obtained automatically by the *quantitative runtime monitoring* or manually evaluated by engineers during simulations of the system verification. Qualitative statements will be assumed as reliable and sound for real-world situations if the behavior of autonomous vehicle systems has been verified and validated in these situations.

The application of qualitative statements to the unknown real world is not sound. Autonomous vehicle systems may exhibit diverging and unsafe behavior in real-world situations even if similar situations have been verified in the simulations of the system verification. The autonomous vehicle system has to be verified in these particular systems.

During operation in the real world, the scopes of qualitative statements have to be evaluated in all encountered situations in order to ensure the safety of the autonomous vehicle systems. The *quantitative monitoring layer* establishes the *tested situation knowledge* with known and verified *abstract situations* (cf. Fig. 5.1). The validity of qualitative statements for the autonomous vehicle systems have been verified and validated in these known and verified *abstract situations* in simulations of the system verification (cf. Section 3.6). The *tested situation knowledge* can be used during operation

in the real world to evaluate the scope of qualitative statements for the autonomous vehicle systems.
The components *situation monitor* and *situation oracle* compare and record the *abstract situations* which are processed by the *input abstraction* (cf. Fig. 5.1). During operation, encountered *abstract situations* are compared with the knowledge of known and verified *abstract situations* from the simulations. Unknown *abstract situations* are recorded in order to address these situations by additional improvements of the autonomous vehicle systems.

5.5.1. Situation Monitor and Situation Knowledge

The *situation monitor* is responsible for the data flow within the *quantitative monitoring layer* and the *tested situation knoweldge*(cf. Fig. 5.1). At initialization time, the situation monitor ensures that the *tested situation knoweldge* contains solely unique *abstract situations* and discards any duplicated situations. The current *abstract situation* from the *input abstraction* and the *abstract situations* of the *tested situation knowledge* are provided to *situation oracle* for comparison. The *situation oracle*i is described in the following section.

5.5.2. Situation Oracle

The *situation oracle* compares each *abstract situation* from the *input abstraction* with the known and verified *abstract situations* in the *tested situation knowledge* (cf. Fig. 5.1). Any encountered unknown *abstract situations* are logged for further analysis in the system evolution (cf. Section 3.1.2). Similar to the *conformity oracle* (cf. Section 5.4.2), the comparison results of the *situation oracle* can be used to initiate safety measures and to immediately mitigate the emerging increased risks in the unknown and unverified situations.
The comparison of *abstract situations* by the *situation oracle* commonly requires the explicit implementation of the equivalence functions. Common equivalence functions, e.g., for integer or float arithmetics, are not applicable for the complex data types of the abstract representation (cf Section 6.2.2.2.1). For each complex data type of the abstract representation, a corresponding equivalence function has to be implemented. In some use cases, the implementation may incorporate probability distributions under consideration of uncertainties.
The definition and implementation of the runtime monitoring framework and its components are presented in the following chapter. Chapter 6 also describes the application of the *qualitative runtime monitoring* as oracle and the training of the *tested situation knoweldge* in simulations of the system verification. The runtime monitoring of autonomous vehicle systems during operation in the real world is present in Chapter 7.

6. Monitor Engineering and Training

The components of the runtime monitoring framework (cf. Chapter 5) are implemented based on artifacts from the development of autonomous vehicle systems. The *qualitative runtime monitoring* depends on the requirements and specifications of autonomous vehicle systems while the *quantitative runtime monitoring* depends on the test scenarios and test cases for the system verification in simulations. As shown in Fig. 6.1, requirements from the system specification are used to define the *input abstraction*, *output abstraction*, *abstract function*, and *conformity oracle* in order to verify the safe and correct behavior of autonomous vehicle systems. For the *quantitative runtime monitoring*, the recorded *abstract situations* from the execution of test cases in simulations are used as knowledge for the runtime monitoring of autonomous vehicle systems during operation in the real world.

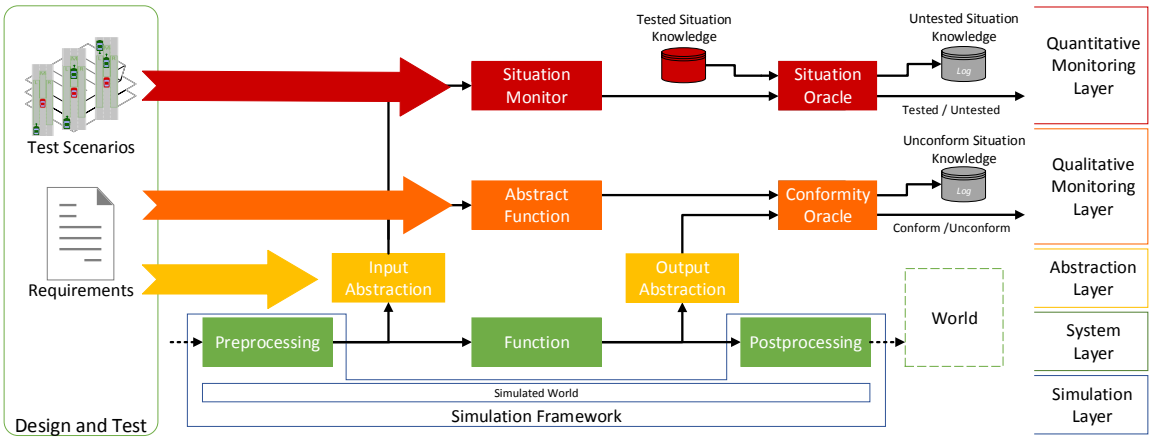


Figure 6.1.: Generation of Runtime Monitoring Components.

The development of runtime monitoring components is described in Section 6.2. In Section 6.3, the usage of runtime monitoring framework in simulations is presented. In the next section, the formal foundation of the runtime monitoring is introduced.

6.1. Formal Representation of the Runtime Monitoring

The formal representation defines the common core functions and characteristic properties of the runtime monitoring framework, its components, and its interfaces allowing the implementation of the runtime monitoring framework for different autonomous vehicle

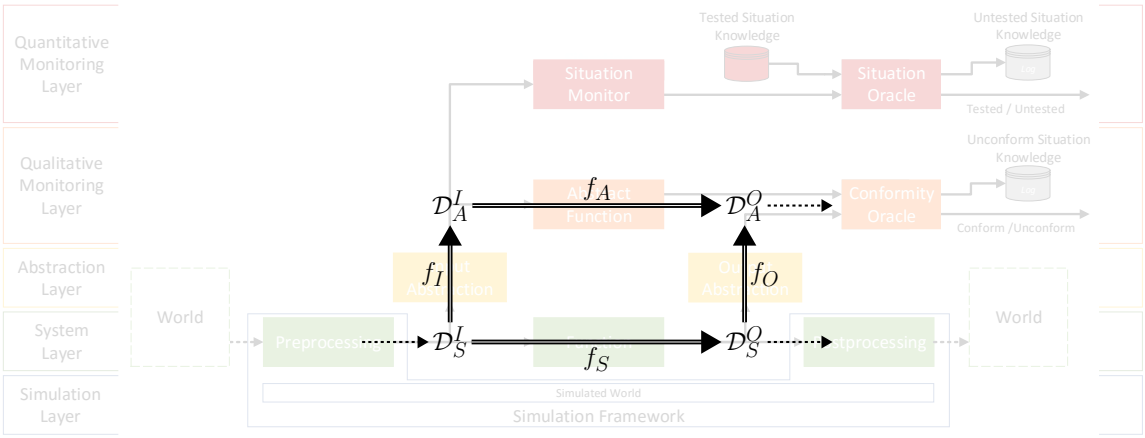


Figure 6.2.: Mathematical representation of the runtime monitoring.

systems. As shown in Fig. 6.2, the formal representation defines runtime monitoring framework by its domains and its functions. The functions f_I, f_S, f_O, and $_A$ correspond to the components *input abstraction*, *function*, *output abstract*, and *abstract function* in the architecture of the runtime monitoring framework (cf. Fig. 5.1). The domains represent sets of data objects which are used by the runtime monitoring framework for the reasoning about the qualitative properties of autonomous vehicle systems and their quantitative scopes. The definition of the monitoring domains is described inSection 6.1.1 and the monitoring functions is presented in Section 6.1.2.
The formal representation enables the definition of a correctness condition for the system behavior and a soundness property for the runtime monitoring framework based on the functions and independent from the concrete autonomous vehicle systems (cf. Fig. 6.2). Any implementation of the runtime monitoring framework has to satisfy these properties. Violations of these properties compromise the overall validity of the runtime monitoring and its results. The correctness condition is presented in Section 6.1.3 while the soundness property is described in Section 6.1.4

6.1.1. Domains

Four domains $\mathcal{D}_S^I$, $\mathcal{D}_S^O$, $\mathcal{D}_A^I$, $\mathcal{D}_A^O$ are defined for the runtime monitoring (cf. Fig. 6.2). The four domains are defined as followed:

$\mathcal{D}_S^I$ depicts the input domain for the *function* of the autonomous vehicle system (cf. Fig. 5.1). The domain $\mathcal{D}_S^I$ is induced by the data types, their properties, and associations between them which are used by the autonomous vehicle system to describe and reason about itself and the vehicle's environment. The input domain $\mathcal{D}_S^I$ for the lane change assistant is shown in Fig. 3.9.

$\mathcal{D}_S^O$ is the output domain of the *function* (cf. Fig. 5.1). It represents the set of actions which the monitored *function* may process as output. The output domain $\mathcal{D}_S^O$ for the lane change assistant is depict by Fig. 3.13.

$\mathcal{D}_A^I$ depicts the input domain of the runtime monitoring framework. As co-domain of the *abstraction* (cf. Fig. 5.1), $\mathcal{D}_A^I$ represents the abstract representation of the domain $\mathcal{D}_S^I$. The domain $\mathcal{D}_A^I$ is defined by a typed first-order logic (cf. Section 2.4) from the requirements of each particular autonomous vehicle system. The definition of domain $\mathcal{D}_A^I$ is described in Section 6.2.2.2. For the lane change assistant, the abstract state of the autonomous vehicle systems and their environments (*abstract situation*) is represented by the domain $\mathcal{D}_A^I$ (cf. Fig. A.2).

$\mathcal{D}_A^O$ depicts the abstract representation of all possible outputs of the *function* (cf. Fig. 5.1) and the co-domain of the *abstract function* and *output abstraction*. The domain encompasses all actions for an autonomous vehicle system which are considered by the system's requirements. Similar to the domain $\mathcal{D}_A^I$, the domain $\mathcal{D}_A^O$ is defined by a typed first-order logic from the system requirements (cf. Section 6.2.2.2). For the lane change assistant, the possible target lanes for lane changes define the domain $\mathcal{D}_A^O$.

The domains $\mathcal{D}_S^I$, $\mathcal{D}_S^O \subseteq \mathcal{D}_S$ are given by the autonomous vehicle systems. Domain $\mathcal{D}_S$ encapsulates all objects for the data types of the autonomous vehicle systems. The domains $\mathcal{D}_A^I, \mathcal{D}_A^O \subseteq \mathcal{D}_A$ have to be defined for the runtime monitoring based on the requirements of each particular autonomous vehicle system. Similar to $\mathcal{D}_S$, the domain $\mathcal{D}_A$ encapsulates all data objects used by the runtime monitoring of an autonomous vehicle system.
The domains $\mathcal{D}_S^I$,$\mathcal{D}_S^O \subseteq \mathcal{D}_S$ represent the interfaces between the runtime monitoring framework and autonomous vehicle systems. Both domains have to be selected and formalized in conjunction with the definition of the domains $\mathcal{D}_A^I, \mathcal{D}_A^O \subseteq \mathcal{D}_A$. The system data is subject to the development of the autonomous vehicle system, but multiple data interfaces within autonomous vehicle systems can be selected and aggregated to represent the domains $\mathcal{D}_S$. The overall set of data elements by these interfaces has to be adequate for the *input abstraction* and *output abstraction* to process all necessary data elements for the domains $\mathcal{D}_A$. Otherwise, not all data elements of the abstract representations $\mathcal{D}_A$ can be processed in order to evaluate the requirements of the autonomous vehicle systems entirely.
Besides the four domains for the runtime monitoring framework, four functions are included in the formalization of runtime monitoring (cf. Fig. 6.2). The four functions are described in the following section.

6.1.2. Functions

For the *qualitative runtime monitoring* , three functions are defined (cf. Fig. 6.2). These three functions f_I, f_A, and f_O correspond to the components of the framework (cf. Fig. 5.1); function f_I represents the *input abstraction*, function f_A corresponds to the *abstract function*, and function f_O represents the *output abstraction*. Additional to these three functions for the runtime monitoring framework, the function f_S represents the original *function* of the autonomous vehicle system (cf. Fig. 5.1). The functions are defined as followed:

$$f_I : \mathcal{D}_S^I \rightarrow \mathcal{D}_A^I \tag{6.1}$$
$$f_A : \mathcal{D}_A^I \rightarrow 2^{\mathcal{D}_A^O} \tag{6.2}$$
$$f_O : \mathcal{D}_S^O \rightarrow \mathcal{D}_A^O \tag{6.3}$$
$$f_S : \mathcal{D}_S^I \rightarrow \mathcal{D}_S^O \tag{6.4}$$

The *abstract function* f_A differentiates from the other functions f_I, f_O, and f_S. The output of *abstract function* f_A is a set of elements from domain $\mathcal{D}_A^O$ — the safe actions for the autonomous vehicle systems in the (current) *abstract situation* from domain $\mathcal{D}_A^I$. In general, none of the functions is injective (cf. Eqs. (6.1) to (6.4)). The functions f_I and f_O are abstractions; multiple elements of their input domains $\mathcal{D}_S^I$ resp. $\mathcal{D}_S^O$ are mapped to identical elements of their output domains $\mathcal{D}_A^I$ resp. $\mathcal{D}_A^O$. Functions f_A and f_S process identical system actions for different inputs from the domains $\mathcal{D}_S^I$ resp. $\mathcal{D}_A^I$. For example, the lane change assistant (f_S) initiates lane changes to the left neighbor lane (*output* in $\mathcal{D}_S^O$) in various different traffic situations (*inputs* from $\mathcal{D}_S^I$). The identical applies to the *abstract function* f_A. The *abstract function* just processes the set of actions from $\mathcal{D}_A^O$ identically for various *abstract situations* from domain $\mathcal{D}_A^I$.
Ideally, all functions Eqs. (6.1) to (6.4) are surjective. Every element of the functions' co-domains can be processed. However, cases arise in which the functions Eqs. (6.1) to (6.4) will not be surjective. Requirements may define unrealistic objects leading to the fuzzy definition of functions' co-domains. Functions are unlikely to process these fuzzy objects b based on realistic input from the autonomous vehicle system. Other cases are insufficient interfaces to the autonomous vehicle systems. The available data by the autonomous vehicle systems may be insufficient to process the complete co-domains.
Implementations of functions Eqs. (6.1) and (6.3) for the *abstraction* of system input and system output need not to be deterministic. Each processing cycle, the functions f_I and f_O transform an element from the input domain into an element of the output domain. For the lane change assistant, identical situation representations from the autonomous vehicle system ($\mathcal{D}_S^I$) have to be transformed by the *input abstraction* (function f_I) to the same *abstract situation* ($\mathcal{D}_A^I$). These transformations are independent of the history of the system inputs ($\mathcal{D}_S^I$). However, internal states within transformations may vary for identical inputs in different processing cycles leading to nondeterministic transformations (cf. [Flo67]). The same applies to the *output abstraction* (function f_O). For the runtime monitoring, the data history during operation is given by the sequence of states in the domain $\mathcal{D}_A^I$.
The system function f_S is nondeterministic. Different system actions may be selected for identical inputs. The selection of system actions in the behavior planning requires non-deterministic choices if the reward for multiple paths in the belief tree is the same (cf. Section 3.3.1.3).
Opposed to the system function f_S, the *abstract function* f_A of the runtime monitoring framework does not process one final system action but a set of correct and safe system actions $2^{\mathcal{D}_A^O}$. The processing of the *abstract function* f_A can be nondeterministic. The

internal states of the *abstract function* f_A may be inconsistent for identical input situations in different processing cycles even though the *abstract function* f_A processes identical sets of system actions for identical input situations. The *abstract function* f_A can even process diverging outputs for identical input situations if its internal states rely on information from the infinite history of inputs.
In addition to the definition of the domains and functions, a condition for correct and safe system behavior is defined. This correctness condition is presented in the following section.

6.1.3. Correctness Condition

The processing of system *function* f_S will be safe and correct if the Eq. (6.5) is satisfied. The processing of inputs about the system and environment state $x \in \mathcal{D}_S^I$ by the system *function* f_S and *output abstraction* f_O have to match the results of the processing by *input abstraction* f_I and *abstract function* f_A for identical inputs x. The evaluation of Eq. (6.5) is subject to each autonomous vehicle system and is implemented by the *conformity oracle*.

$$\forall x \in \mathcal{D}_S^I : f_O\left(f_S\left(x\right)\right) \in f_A\left(f_I\left(x\right)\right) \tag{6.5}$$

The *conformity oracle* is implemented as a function over $\mathcal{D}_A^O$. It evaluates if the output by the *output abstraction* f_O is included in the power set of the *abstract function* f_A (cf. Eq. (6.6)). The output of the *conformity oracle* resides in $\{\text{True}, \text{False}\}$ and indicates the compliance of the system *function* f_S to its qualitative properties. In Fig. 6.2, the *conformity oracle* is omitted for simplicity reasons.

$$f_C : \mathcal{D}_A^O \times 2^{\mathcal{D}_A^O} \rightarrow \{\text{True}, \text{False}\} \tag{6.6}$$

All statements by the runtime monitoring framework about correct and safe system behavior require the runtime monitoring to be sound. The following section introduces a soundness property for the runtime monitoring framework. This soundness property addresses the semantic identity of the semantic interpretations in the system domains and the runtime monitoring domains. Otherwise, the validity of statements about correct and safe system behavior cannot be guaranteed.

6.1.4. Soundness property

The abstraction by the runtime monitoring framework has a significant impact on the validity of statements about the correctness and safety for autonomous vehicle systems. Interpretations in the semantic domain $\mathcal{S}$ have to be maintained. The semantic domain $\mathcal{S}$ can be infinite.
Two functions c_S and c_A can be defined; function c_S maps elements from the domain $\mathcal{D}_S^I$ to the semantic domain $\mathcal{S}$ and c_A maps elements from the domain $\mathcal{D}_A^I$ to the semantic domain $\mathcal{S}$.

$$c_S : \mathcal{D}_S^I \rightarrow \mathcal{S} \tag{6.7}$$
$$c_A : \mathcal{D}_A^I \rightarrow \mathcal{S} \tag{6.8}$$

An example of a semantic domain is the safety of autonomous vehicle systems in traffic situations. The behavior of autonomous vehicle systems can be judged as safe or unsafe for encountered real-world traffic situations (cf. Section 3.8.1). In this example, the interpretations of the semantic domain $\mathcal{S}$ are $\mathcal{S} = \{\text{Safe}, \text{Unsafe}\}$.
Functions c_S and c_A need not to be injective nor subjective. The interpretations in the semantic domain $\mathcal{S}$ can be vast or even infinite but not all interpretations have to be considered by functions c_S and c_A. Multiple objects of domains $\mathcal{D}_S^I$ resp. $\mathcal{D}_A^I$ may have identical semantic interpretations in $\mathcal{S}$.
The validities of statements require the abstraction by functions f_I and f_O to maintain the same interpretation in the semantic domain $\mathcal{S}$. The functions f_I and f_O must not abstract two different elements from $\mathcal{D}_S^I$ resp. $\mathcal{D}_S^O$ which relate to different elements in the semantic domain $\mathcal{S}$ into objects in the domains $\mathcal{D}_A^I$ resp. $\mathcal{D}_A^O$ with identical interpretation in $\mathcal{S}$. The same applies to the opposite case — two elements from $\mathcal{D}_S^I$ resp. $\mathcal{D}_S^O$ with identical interpretation in $\mathcal{S}$ must not have diverging interpretations after application of f_I resp. f_O. The Eq. (6.9) defines the consistency of the semantic interpretation for $f \in \{f_I, f_O\}$. It must hold for f_I and f_O at all times.

$$\forall s_1, s_2 \in \mathcal{D}_S^I \text{ s.t. } a_1 = f(s_1) \wedge a_2 = f(s_2) \wedge c_A(a_1) = c_A(a_2) \Leftrightarrow c_S(s_1) = c_S(s_2) \tag{6.9}$$

For the *quantitative runtime monitoring*, no further functions or domains have to be defined. The *quantitative runtime monitoring* solely records and compares instances of the abstract representation D_A^I. Therefore, only the relation to the comparison of abstract representations has to be defined. The implementation of the recording is described in Section 6.2.3.5.
The formalization of the runtime monitoring forms the foundation for the implementation of the runtime monitoring framework. Domains and functions of the runtime monitoring framework have to be defined based on the requirements for the autonomous vehicle systems. The development process of the runtime monitoring framework is described in the following section

6.2. Development of Runtime Monitors

The development of the runtime monitoring framework is performed concurrently to the development of autonomous vehicle systems because the runtime monitoring framework depends on interfaces and requirements of each particular autonomous vehicle system. These requirements have to describe correct and safe resp. incorrect and unsafe system behavior sufficiently.

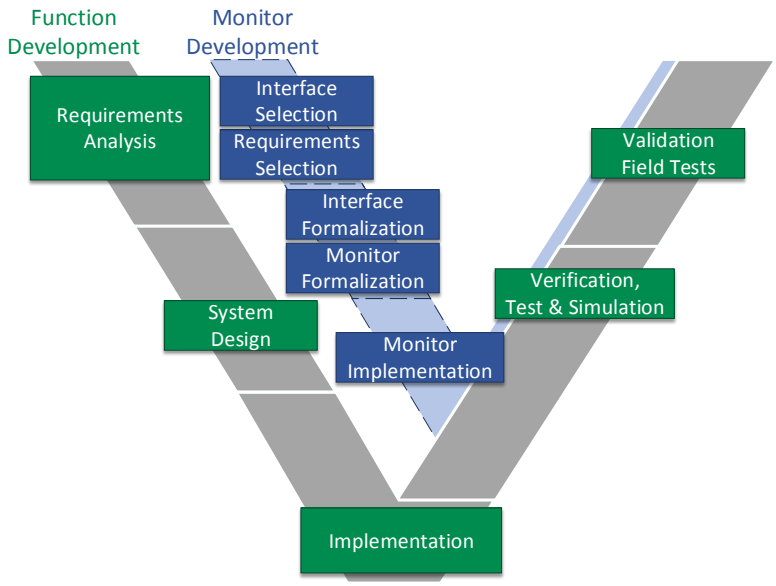


Figure 6.3.: Integration of the monitor development into the development of autonomous vehicle functions.

As shown in Fig. 6.3, the development of runtime monitors starts with the selection of system interfaces and requirements in collaboration with requirements analysis for autonomous vehicle systems. These requirements are commonly defined in natural language and have to be analyzed and formalized as formal logic in order to define the domains $\mathcal{D}_A^I$, $\mathcal{D}_A^O$ and functions f_I, f_A, f_O of the runtime monitoring. As result of the *monitor formalization*, each considered requirement is expressed by one or more formulas. These formulas are implemented as *runtime monitors* (cf. Section 2.3.1). Sections 6.2.1 to 6.2.3 describe the development activities for runtime monitoring framework in more detail.

6.2.1. Selection of Interface and Requirements

Relevant requirements for the runtime monitoring are selected by domain and safety experts from the overall set of requirements for autonomous vehicle systems in the requirements analysis (cf. Section 3.2). These requirements have to describe the correct and safe resp. incorrect and unsafe system behavior of autonomous vehicle systems which shall be supervised by the runtime monitoring framework sufficiently.

Safety-related requirements are of special interested for the runtime monitoring. Safety requirements define constraints on the basic functionality in order to maintain the system's resp. function's correctness and safety (cf. Sections 3.2.1 and 3.2.3). Compared to traditional E/E systems in the automotive domain, requirements for autonomous vehicle systems have to address the common internal properties and conditions of these systems and their environments. Following the open-world assumption, the requirements

inherent an uncertainty to sufficiently represent the complex nature of the real world and all its objects and properties (cf. Section 3.8.1).
Additional to the selection of relevant safety requirements, interfaces of autonomous vehicle systems are identified and selected as data sources for the runtime monitoring. A sufficient definition of these interfaces in the early stages of the development supports the independence between the further development of the autonomous vehicle systems and the runtime monitoring framework. In an ideal case, this dependency reaches till their integration of implementations with each other for verification in simulation and validation during operation in the real world (cf. Fig. 6.3).
However, all development decisions regarding requirements, system behavior, and interfaces in the further development phase of autonomous vehicle systems have to be considered in the development of the runtime monitoring framework.
For the implementation of the runtime monitoring framework, interfaces between system and framework as well as the monitoring components have to be formalized. This formalization is described in Section 6.2.2. The implementation of each monitoring component is presented in Section 6.2.3.

6.2.2. Formalization of Interfaces and Requirements

The definition of domains and functions for the runtime monitoring framework (cf. Fig. 6.2) requires the identification of objects, types, and their relations which are present in considered requirements of the autonomous vehicle systems. Requirements are commonly formulated in natural language leading to the possibility of ambiguous formulations. Such ambiguous formulations make it difficult to correctly identify contained objects, types, and their relations (cf. [Ber10; Kan15]). The implementation of the runtime monitoring framework requires requirements to sufficiently describe relevant conditions for the behavior of autonomous vehicle systems. Domain experts are required to analyze requirements and to enrich them with information which have been omitted in the requirements analysis but which are necessary for the runtime monitoring. This analysis includes implicit domain knowledge which engineers assume in the requirements engineering but do not explicitly express in the requirements.
For the definition of interfaces and runtime monitors, a formalization process is applied to requirements which are considered for the runtime monitoring. Other formalization approaches can be found in [Bec+14; Ber10; Cim+10; Kan15]. In this *monitor formalization*, requirements in natural language are transformed into typed first-order logic (cf. Section 2.4) based on the result from a pattern-based analysis. The pattern-based analysis helps engineers to explicitly formulate previous implicit domain knowledge in the requirements (cf. [ADT14]). The *typed first-order logic* can be directly implemented as software runtime monitors.
As a result of the *monitor formalization*, each safety requirement is expressed by a formula in *typed first-order logic*. These formulas define the types, objects, and relations which constitute the domains $\mathcal{D}_A^I$, $\mathcal{D}_A^O$ and functions f_I, f_A, f_O. Types and objects define the domains of the runtime monitoring $\mathcal{D}_A^I$, $\mathcal{D}_A^O$ while the *abstract function* f_A evaluates

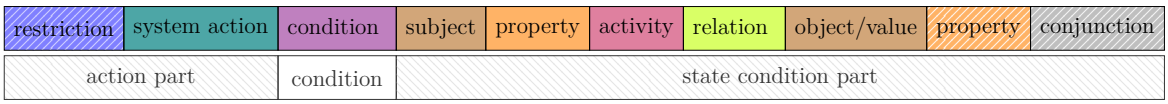


Figure 6.4.: Requirements pattern.

predicates over the elements of these domains in order to reason about the correctness and safety of the system behavior.

> *Example* 6.1. The system shall be able to consider fast objects on the neighbor lane approaching the ego vehicle from behind with at least 5 m/s relative velocity for a lane change to the left neighbor lane.

Example 6.1 displays an extended version of a requirement from the lane change assistant (cf. Section 3.2. In the following sections, Example 6.1 is used as the running example for the description of *monitor formalization*.
The pattern and its application in the *monitor formalization* is described in Section 6.2.2.1. Section 6.2.2.2 presents the definition of types, variables, functions, predicates, and formulas of the typed first-order logic from the analyzed requirements.

6.2.2.1. Pattern-based Analysis of System Requirements

For valid and comprehensive results of the runtime monitoring framework, system requirements for autonomous vehicle systems have to describe these systems as well as their environments sufficiently precise. Nevertheless, engineers often implicitly assume domain knowledge for the definition of requirements in the requirement engineering. Any implicitly assumed domain knowledge is difficult — if not impossible — to be correctly implemented by the algorithms of the runtime monitoring framework.
A pattern-based approach similar to [Bit01; Kan15; Mac+14] is used for the formalization and structuring of requirements in order to address the problem of imprecise and ambiguous requirements. The requirements pattern (cf. Fig. 6.4) requires engineers to explicitly expressed all necessary domain knowledge in the requirements for the runtime monitoring. For example, properties of objects such as distance or position are seldom explicitly mentioned in requirements but assumed from the contexts of these requirements. Therefore, the requirements pattern reduces the potential ambiguity of requirements and failures in the implementation of the runtime monitoring framework. Requirements considered for the runtime monitoring have to formulated that their phrases match the categories of the pattern in order to define objects, types, and their relations for the domains and functions of the runtime monitoring. This restriction for the formulation of sentences leads to the consideration of pattern-based formalization approaches as restricted natural language (cf. [THH06]).
The high level structure of the requirement pattern reflects the relationship of system inputs and outputs which the runtime monitor framework is supervising. The pattern format consists of a condition about the current state of the system and its environment (*state condition part*) and a corresponding implication of a system action (*action part*).

Table 6.1.: Possible wording for relation in natural language requirements.

on	next	behind	before	parallel
less (than)	more (than)	equal to	within / in	

In formal logic, this would relate to *statecondition* $\rightarrow$ *action*. In Fig. 6.4, the *action part* corresponds to the part of the requirement prior to the *condition* (cf. "No lane change" in Example 6.1). The *state condition part* corresponds to the remain part of the requirement and is introduced by the *condition if*. The *if-condition* separates *state condition part* and *action part* and implicitly represents the implication between them. The *state condition part* and *action part* of the requirement pattern is described in more detail in the following sections.

6.2.2.1.1. State Condition Part

The *state condition part* commence with an object (the condition's *subject*) and its *property*. The object can be the system, a single real-world object, e.g., a bicycle, but also a variable for a set of real-world elements, like, e.g., vehicles. The property defines a particular property of these objects about which is reasoned in the condition.
The *activity* follows the *object property*. The *activity* refers to the verb of the requirement which commonly describes an action or state of the subject. For example, the description of the object's property's current state (*to be*) is a fundamental activity element. Other activities include, e.g., to move, to drive, or to overturn.
The *activity* is followed by a condition in relation to another object, in relation to a explicit property value, or a relation to an explicit property value of another object. A incomplete list of possible relations is presented in Table 6.1. The related object is represented in Fig. 6.4 by the *object* category while or the concrete property value of the subject is depict by the *data type value* category in Fig. 6.4.
The application of the pattern from Fig. 6.4) for Example 6.1 is shown in Fig. 6.5. It is apparent that several modifications have been necessary. The pattern defines one condition for one particular property of an object (subject). Requirements, like Example 6.1, may include conditions for multiple object properties. Each property condition in a requirement has to be separately expressed in order to reason about the object properties in the formal logic without any ambiguity and side effects. All distinctive conditions are aggregated in the *state condition part* by *conjunctions* in order to correctly represent the initial requirement.
The Example 6.1 contains two conditions. The first two conditions in Fig. 6.5 reason about a object property (vehicle position) in relation to other objects. The first condition reasons about the position of a vehicle in relation (on) to the road lane (left neighbor lane) while the second conditions reasons about the position of the vehicle in relation (behind) to the automated ego vehicle. The third condition evaluates a the object property (vehicle relative velocity) in relation to a defined property value ($5\,\mathrm{m\,s^{-1}}$) (cf. third line of Fig. 6.5).

No	lane change to the left lane	if	vehicle	position	is	on	the left neighbor lane	and
Restriction	system action	condition	subject	property	activity	relation	object (place)	conjunction

No	lane change to the left lane	if	vehicle	position	is	behind	the ego vehicle	and
Restriction	system action	condition	subject	property	activity	relation	object (place)	conjunction

No	lane change to the left lane	if	vehicle	relative velocity	is	more than	5 m/s.
Restriction	system action	condition	subject	property	activity	relation	data type value

Figure 6.5.: Application of the requirement pattern for Example 6.1.

The second category *property* has been omitted for (cf. Example 6.1 because the position and relative velocity of the vehicle are evaluate in relation to other objects and not the objects' properties (cf. Fig. 6.5).

Relations in requirements can not only relate to one object but also a list of objects. Our requirements pattern can address this description in natural language requirements by transforming conditions with sets of related objects into multiple atomic conditions with each just having one related object. These atomic conditions are combined by *and* or *or* to represent the identical state condition. This way the number of atomic conditions for a state condition part increases but each atomic condition has only one relation between two objects and their properties (data values). Multiple different state conditions can be combined if they relate to the same set of actions. For example, in the second line of Fig. 6.5 the action part is grayed while an *and-conjunction* is added to the first part.

For the results displayed in Fig. 6.5, additional changes have been introduced in order to further reduce the ambiguity of the requirement (cf. Example 6.1). The activity verb of the requirement has been changed from "to consider" to "to be" has been added as corresponding system action in order to avoid the ambiguity of "to consider". The atomic condition addressing the "relative velocity" has been reordered and "relative velocity" has been explicitly defined as object property rather than as data type in order to properly fit the pattern.

The second part of the pattern about required or restricted system actions is described in the following section.

6.2.2.1.2. Action Part

The *action part* of the pattern primarily consists of the system actions which refer to the possible output of the system function f_S (cf. Fig. 6.4) . As the runtime monitoring focuses on safety related requirements, the *action part* may require mandatory actions or restrict the execution of actions by the system function f_S under the specific conditions of the *state condition part*. The pattern introduces the optional element *restriction* in the *action part* for the restriction of system actions (cf. Fig. 6.4). Multiple system actions considered with one *state condition part* can be conjuncted by *and* or *or* — similar to the conjunction used for multiple atomic conditions. More complex system actions have to be analyzed and formalized in the same way as the *state condition part*.

In the application of the pattern to Example 6.1 (cf. Fig. 6.5), the initial term "to consider" from Example 6.1 is substituted by the restriction of an lane change ("No lane change"). The *action part* of the pattern has to relate to actions or behavior of the system. The initial term "to consider" in Example 6.1 is ambiguous and does not relate to any action by the lane change assistant. The action "lane change" represent the implicit meaning of the term "to consider" in Example 6.1; to check the conditions for the vehicle environment prior to changing to the left lane. As the Example 6.1 represents an exclusion of a lane change in the specific conditions, the system action has to also be restricted ("No lane change to the left lane").

The categorization of requirements by the pattern (cf. Fig. 6.4) enables the representation of requirements in formal logic. Objects, types, and relation defining the domains $\mathcal{D}_A^I$, and $\mathcal{D}_A^O$ can be identified. Relations between objects and types are used in the definition of functions f_I, f_A, and f_O. The definition of requirements in formal logic for the runtime monitoring of autonomous vehicle systems is described in the following section.

6.2.2.2. Definition of Typed first-order Logic

For the definition of domains $\mathcal{D}_A^I$, $\mathcal{D}_A^O$ and functions f_I, f_A, f_O, all patterned requirements (cf. Fig. 6.4) are transformed into typed first-order logic (cf. Section 2.4). The formal description as formulas enables the identification of types, objects, and relations defining these domains and functions for the runtime monitoring framework. The typed first-order logic has been introduced first in [BHS07]. A type hierarchy is defined which allows the binding of variables, functions, and relations to specific types (cf. Section 2.4.1). The type hierarchy is similar to the data structures in [Cim+10; Kan15]. Further details about this typed first-order logic are given in Section 2.4.

First-Order logic is necessary because runtime monitoring of autonomous vehicle systems requires the reasoning about multiple (all) objects of a particular type. Considering the open-world assumption (cf. Section 3.8.1), the quantification over objects is essential in order to correctly address the infinite sets of objects in systems' real-world environments. For example, the safety of the lane change assistant depends on the positioning of all vehicles in the vicinity of the autonomous vehicle. Propositional logic cannot reason about individual objects, sets of objects, and relations among them in sufficient detail for the runtime monitoring (cf. [BHS07; BKV13]). Propositional logic could only evaluate if specific zones in the vicinity of the automated ego vehicle are occupied but would not allow any reasoning about the individual behavior of traffic participants. Knowledge about interactions between other traffic participants and the autonomous vehicle systems is vital for the improvement of autonomous vehicle systems based on the runtime monitoring results. The modeling of world traffic situations in test scenarios and test cases requires the explicit description of individual real-world objects in the runtime monitoring results (cf. Section 7.2.2.

The following sections describe the definition of types, variables, functions, predicates, and formulas of the typed first-order logic for the runtime monitoring from patterned requirements (cf. Section 2.4).

6.2.2.2.1. Types and Co-Domains

For the typed first-order logic of the runtime monitoring, types have to be identified for the objects in the requirements in order to reason about the real world on an abstract level. A type defines a class of real-world objects which share the similar properties (cf. Section 6.2.2.2.1). The definition of types includes the definition of their value ranges — the set of abstract values which objects of this type can adopt in the formal logic. The types identified in the requirements define the domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$ for the runtime monitoring framework (cf. Fig. 6.2).

Each requirement commonly defines multiple types for the *typed first-order logic*. As shown in Fig. 6.4, types are defined for each element of the pattern categories *subject*, *object*, as well as their *properties*. Example 6.1) defines the types *vehicle*, *lane*, *relative velocity*, *(lane) position*, and *(relative) position* (cf. Fig. 6.5). Any type may occur in multiple requirements.

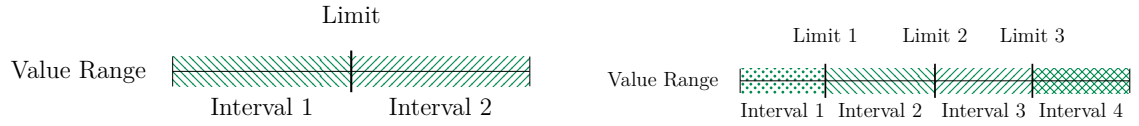
Types define ranges of possible values. For the runtime monitoring framework, ranges of values for types of the domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$ are defined by the objects and data values within predicates of the requirements (cf. Section 2.4.2). Complex objects and data types have to be distinguished for the typing of the first-order logic. Complex objects refer to real-world objects while data values which define limits on measurable types. An order relation commonly sorts the values of a data type. The order relation is not mandatory for complex objects which represent real-world objects. Sets of complex objects and value ranges of data types need not to be finite and denumerable. Standard data types with infinite domains are, e.g., integer, float, and real. For the lane change assistant, the set of possible objects for the type *vehicle* is infinite but denumerable.

Objects in predicates (cf. category *object* in Fig. 6.4) refer to explicit *object instances*. An *object instances* represents one particular object in the set of possible object for the particular type. In Fig. 6.5, the terms *left neighbor lane* and *ego vehicle* in the category *object* refer to real world objects; *left neighbor lane* refers to a specific real world lane and *ego vehicle* refers to the automated (ego) vehicle. Each object instance is denoted by unique identifier or uniquely descriptive adjectives, e.g., *left lane* or *right lane* (cf. Fig. 6.5).

In the first-order logic, each specific object is represented as constant — a nullary function symbol. For the runtime monitoring framework, these object instances represent a subset of possible values for the corresponding type. There may not exist any universally defined order relation for objects of such types.

For the lane change assistant, the *ego vehicle* is an explicit *object instance* for the type *Vehicle* but there is a infinite number of other vehicles which have to be considered for the runtime monitoring. The type *lane* only considers the relevant lanes for intimidate lane changes, the left and right adjacent lanes, and the (ego) lane on which the automated vehicle drives as object instances: $\mathbb{D}_{\text{Lane}} = \{\text{Left}, \text{Ego}, \text{Right}\}$. The three lanes define the complete range of values for the type *lane* while the object *ego vehicle* represents a subset of the possible values for the type *vehicle*.

For the runtime monitoring, additional types are introduced alongside existing primitive data types, e.g., integer, float, or real, in order to describe abstract value ranges of object



(a) Definition of abstract values for a single limit.

(b) Definition of abstract values for mutiple limits.

Figure 6.6.: Definition of abstract values for types.

properties. Value limits of predicates commonly refer to number-valued types which can be measured by the autonomous vehicle system (cf. *relative velocity* in Example 6.1). These types commonly define an order relation on their values, e.g., integer, float, or real. Such order relations allow the definition of abstract intervals from the number-valued types as abstract values for the corresponding types in $\mathcal{D}_A^I$ resp. $\mathcal{D}_A^O$. For each limit, two intervals are defined: one interval matching the limit of the predicate and one interval violating the predicates limit (cf. Fig. 6.6a). Each interval is denoted in the runtime monitoring framework as a unique object by its unique identifier. In the case predicates define various limits for the same number-valued type, multiple abstract values are defined for this type in $\mathcal{D}_A^I$ resp. $\mathcal{D}_A^O$ in order to correctly represent the intersection of to match the intersection of intervals over the limits (cf. Fig. 6.6b).
Equation (6.11) shows the definition of abstract values for the relative velocity in Example 6.1). For the predicate "more than 5 m/s" of Example 6.1) (cf. Fig. 6.5), one interval *LOWER* is defined for values lower than 5 m/s and one interval *HIGHER* is defined for values equal or higher than 5 m/s. Values for the relative velocity are transformed accordingly to the abstract values *LOWER* resp. *HIGHER* (cf. function f_I^{Vel} in Eq. (6.11)).

$$\mathbb{D}_{RelativeVelocity} = \{\text{LOWER}, \text{HIGHER}\} \tag{6.10}$$
$$f_I^{Vel} : \mathbb{R} \rightarrow \mathbb{D}_{RelativeVelocity} \tag{6.11}$$
$$f_I^{Vel}(-0.2) = \text{LOWER} \tag{6.12}$$
$$f_I^{Vel}(4.1) = \text{LOWER} \tag{6.13}$$
$$f_I^{Vel}(5.0) = \text{HIGHER} \tag{6.14}$$
$$f_I^{Vel}(120.0) = \text{HIGHER} \tag{6.15}$$

For the evaluation of predicates (cf. Section 2.4.2) by the runtime monitoring framework, the order relation for number-valued types from the domains $\mathcal{D}_S$ has to be maintained for the range of abstract values of corresponding types in $\mathcal{D}_A$. The order relation on the abstract values of $\mathcal{D}_A$ enables to referencing of subsets of valid values in predicates. Otherwise, predicates referring to value limits cannot be easily evaluated for types with sets of abstract values of 3 or more elements. Otherwise, each abstract value which meets the limit, e.g., 5 m/s, would have to be independently compared to the current abstract value of the evaluated object property.

In case, an additional requirement introduces another limit for the relative velocity, the set of abstract values has to be extended by refinement of affected intervals. Let's say the additional requirement defines the condition "equal or higher than 10 m/s", the abstract value *HIGHER* has to be refined. As shown in Eq. (6.17), the domain $\mathbb{D}_{RelativeVelocity}$ is extended by the abstract value *MID* to represent concrete values between 5 m/s and 10 m/s. The order relation on these abstract values — *LOWER*<*MID*<*HIGHER* — ensures that the requirements can efficiently evaluated in the implementation. All abstract values larger or equal *MID* satisfy the condition "equal or higher than 5 m/s". Abstract values lower and equal *MID* violate the condition "equal or higher than 5 m/s".

$$\mathbb{D}'_{RelativeVelocity} = \{\text{LOWER}, \text{MID}, \text{HIGHER}\} \tag{6.16}$$

$$F_I^{Vel} : \mathbb{R} \rightarrow \mathbb{D}'_{RelativeVelocity} \tag{6.17}$$

$$F_I^{Vel}(-0.2) = \text{LOWER} \tag{6.18}$$

$$F_I^{Vel}(4.1) = \text{LOWER} \tag{6.19}$$

$$F_I^{Vel}(5.0) = \text{MID} \tag{6.20}$$

$$F_I^{Vel}(120.0) = \text{HIGHER} \tag{6.21}$$

Requirements may be an underspecification of the real world. In the real world, objects may occur in the real world situations which do not match any defined type of the requirements resp. the runtime mounting framework. As consequence, an *out-of-range* element is introduced for each type. As processing result of the *input abstraction* f_I or *output abstraction* f_O, a type-specific *out-of-range* element indicates a processing error or the insufficient consideration of the real world by the requirements. The domain for the relative velocity (cf. Eq. (6.11)) would be extended by its type-specific *out-of-range* element OOR to

$$\mathbb{D}^*_{RelativeVelocity} = \{\text{LOWER}, \text{HIGHER}, \text{OOR}\}\,. \tag{6.22}$$

However, the *out-of-range* element OOR need not be utilized for all types. Function f_I^{Vel} (cf. Eq. (6.11)) never processes the element OOR by function f_I^{Vel} as abstract value for the relative velocity. The abstract values *LOWER* and *HIGHER* correspond to two continuous open intervals over the complete real domain $\mathbb{R}$ of the relative velocity.
All identified types for the definition of domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$ are structured in a corresponding type hierarchy for each domain. The type hierarchies are essential for typing entities of the typed first-order logic, e.g., variables, functions, and predicates (cf. Section 2.4). The definition of type hierarchies is described in the following section.

6.2.2.2.2. Domains and Type Hierarchies

For the runtime monitoring, types which have been identified in the requirements are organized in type hierarchies for the domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$ (cf. Fig. 6.2). A *type hierarchy* represents an inheritance hierarchy of types. Each type which has been identified by the pattern-based analysis is encapsulated in the type hierarchies of domains $\mathcal{D}_A^I$ resp. $\mathcal{D}_A^O$.

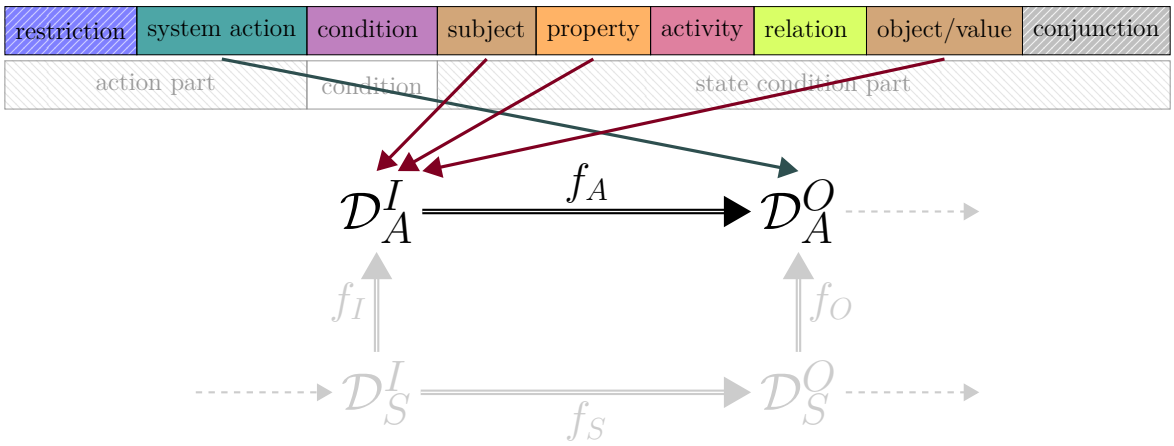


Figure 6.7.: Definition of domains $\mathcal{D}_A^I$, $\mathcal{D}_A^O$ from the requirement pattern.

As shown in Fig. 6.7, the type hierarchy of domain $\mathcal{D}_A^I$ contains types which are defined in the *state conditional parts* of the requirements while the type hierarchy of domain $\mathcal{D}_A^O$ encapsulates types from the *action parts* of these requirements.

Definition 6.1 (Type Hierarchy)**.** *A type hierarchy represents an inheritance hierarchy of (data) types.*

The root of any type hierarchy is always the universal type $\top \in \mathcal{T}_D$ (cf. Section 2.4.1). All other types inherent from the universal type $\top$. Types which have been deduced from requirements for the runtime monitoring framework are dynamic types $\mathcal{T}_D$. Any object process by the *input abstractions* f_I or the output abstract f_O, can only have a dynamic type $t \in \mathcal{T}_D$. The type hierarchy can be extended by abstract types $\mathcal{T}_A$ but no runtime object can have abstract types as their original type (cf. Section 2.4.1). Abstract types group similar types in order to defined more general formulas reasoning about this groups of similar types. For example, the abstract type *traffic participant* would summarize vehicles, trucks, bicycles, and pedestrians. The type hierarchy is complemented by the empty type $\bot \in \mathcal{T}_A$ which is a subtype of all other types in the hierarchy (cf. [BHS07]). Types from domain $\mathcal{D}_A^I$ can be reused for the definition of the domain $\mathcal{D}_A^O$. Requirements for autonomous vehicle system may incorporate identical objects in the description of the *system actions* and corresponding *state conditions*. The domain $\mathcal{D}_A^O$ will be a subset of domain $\mathcal{D}_A^I$ if all types of domain $\mathcal{D}_A^O$ are contained in domain $\mathcal{D}_A^I$: $\mathcal{D}_A^O \subseteq \mathcal{D}_A^I$.
For the lane change assistant, the domain $\mathcal{D}_A^O$ is a real subset of domain $\mathcal{D}_A^I$. Lane changes are interpreted in the abstract domains $\mathcal{D}_A$ based on their target lanes. The corresponding type *lane* in the *action part* (lane change to the left lane) ($\mathcal{D}_A^O$) is identically contained in the *state condition part* ($\mathcal{D}_A^I$) for the position of other vehicles on the road (the left lane) (cf. Fig. 6.5). For complex description of system actions, a pattern-based analysis of the *action part* may lead to more elaborated type hierarchies.
Figure 6.8 displays an exemplary type hierarchy which has been defined based on the results of the pattern-based analysis of Example 6.1 (cf. Fig. 6.5). The root type of the

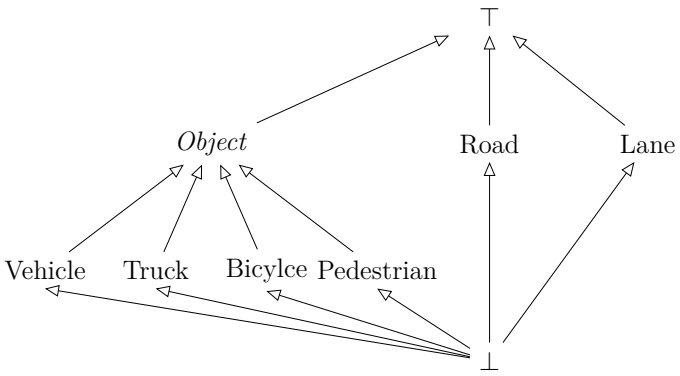


Figure 6.8.: Exemplary type hierarchy.

hierarchy is the universal type $\top$. The type object, road, and lane inherent from the universal type. While types *road* and *lanes* are dynamic types $\{\text{Road}, \text{Lane}\} \in \mathcal{T}_D$, the type *object* is an abstract type $\{\text{Object}\} \in \mathcal{T}_A$ and can only be used in the formulas for safety requirements but can not be assigned to real world objects (cf. Section 2.4.1). The abstract type *Object* encapsulates all types addressing objects that may appear in the vicinity of the automated vehicle. These objects — *Vehicle*, *Truck*, *Bicycle*, and *Pedestrian* — are subtypes of the type *Object*. The types *Vehicle*, *Truck*, *Bicycle*, and *Pedestrian* are all dynamic types $\{\text{Vehicle}, \text{Truck}, \text{Bycicle}, \text{Pedestrian}\} \in \mathcal{T}_D$. The final subtype of the hierarchy is the empty type $\bot$.
The typing of the first-order logic (cf. Section 2.4) enables the restriction of variables, function symbols, and predicates symbols to objects of certain types and to reduces the complexity in the implementation of the runtime monitoring framework. For typing of variables functions, and predicates the typing function α is introduced (cf. Section 2.4.2). The typing function α assign a type to every symbol of V, F, and P and each of their parameters. Variables, function, and predicates are introduced in the following sections.

6.2.2.2.3. Variables

Requirements do not only evaluate properties for single object instances but also reason about properties of sets of objects with identical type. In typed first-order logic, a variable $v \in V$ (cf. Definition 2.36) is defined for each requirements' element of the category *object* (cf. Fig. 6.4), which represents a set of objects of a certain type. In first-order logic, quantifiers $\forall v$ and $\exists v$ are introduced in order to evaluate requirements over all objects of the set v (cf. Section 2.4). The variable v is bind to the particular type of the objects by the typing function α.
For the type of variable as well as parameters of functions and predicates the typing function α is introduced.
In the *state conditional part* of the Example 6.1, the element *vehicle* as subject of the requirement is not an identifier for any explicit object — a constant. The example requirement reasons about the set of all *vehicles* in the perceived environment of the lane change assistant. A variable $y \in V$ is introduced in order to evaluate the requirement for

perceived *vehicles*. Additional to the variable for the vehicle, the Example 6.1 requires the definition of a variable $x \in V$ in order to correctly describe the target lane of the lane change in the *action part* and the *state conditional part*.

6.2.2.2.4. Function Symbols

Unless, primitive data types, e.g., integer, float, and real, types hold a set of properties. For example, the type *vehicle* has properties e.g., *position* and *relative velocity*, in order to describe their position and behavior. Every property represents a *type* which has to be considered in the domains $\mathcal{D}_A$ (cf. Fig. 6.7).
The pattern (cf. Fig. 6.4) defines the class *property* to explicitly express the property of requirements' subjects. Some requirements reason about relations between the properties of objects. Therefore, the pattern also introduces a second *property* category related to the object in the requirement's predicate (cf. Fig. 6.4). for Example 6.1, the second property is committed because the subject properties relate to other objects but not their properties (cf. Fig. 6.5). Elements of requirements for both *property* categories have to be considered in the typed first-order logic.
In first-order logic, properties are represented by a set of function symbols F. The number of parameters for the function symbol $f \in F$ depends on the number of related objects. Properties relating to single objects of particular type, e.g., *position* and *dimensions* for vehicles, can be denoted by unary functions $f \in F$. For an object of type z_1, the function $f(z_1) \rightarrow z$ process an object of type z as value for the corresponding property (cf. Definition 2.36). For example, a function $pos(x_{veh}) = y_{pos}; \alpha(x_{veh}) = \text{Vehicle}, \alpha(y_{pos}) = \mathbb{R}^3$ would return the current position vector of a vehicle.
Properties which are measured in relation to other objects are denoted by n-ary functions where n corresponds to the number of objects in relation. For example, the property *relative velocity* of the Example 6.1 describes the velocity of an vehicle in relation to the ego vehicle with the lane change assistant and can be defined as binary function $rVel(x_{veh}, ego) = y_{vel}; \alpha(x_{veh}) = \text{Vehicle}, \alpha(y_{vel}) = \mathcal{D}_{\text{RelativeVelocity}}$. The constant *ego* describes the ego vehicle with the lane change assistant.
The typing of the first-order logic (cf. Section 2.4), allows to restrict the function parameters to defined types z_i. Opposed to untyped first-order logic, functions are solely evaluated for terms of types $t_i \in Term_{z_i'}$ which meet the type of corresponding function parameters $z_i' \sqsubseteq z_i$ and not for all elements of the domain (cf. Definition 2.39). This restriction corresponds to the implementation in object-oriented language (cf. Section 6.2.3) and reduced the complexity for the implementation of the runtime monitoring framework.

6.2.2.2.5. Predicate Symbols

Requirements define predicates. Predicates relate subjects' properties to other objects, their properties or criteria. The requirements patterns identifies these relations by the category *relation* (cf. Fig. 6.4). A list of possible expressions which describe such relations in requirements is given in Table 6.1.

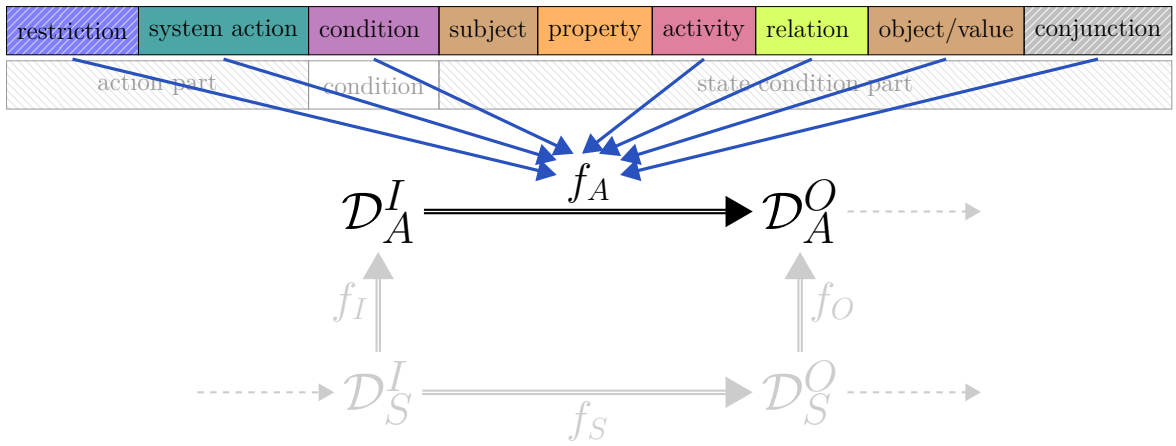


Figure 6.9.: Definition of abstract function f_A from the requirement pattern.

In first-order logic, predicates are denoted by predicate symbols $p(z_1, \ldots, z_n) \in P$ (cf. Definition 2.39). The number of parameter corresponds to the number of related objects, object properties, and constants. Results of predicates reside within $\{\text{true}, \text{false}\} \in \mathit{Boolean}$. Similar to functions, the parameters of predicates are restricted to specific types in order to reduce the complexity of the monitoring algorithm and its implementation. For each parameter a type z_i is defined. Any term $t_i \in Term_{z_i'}$ as parameter has to meet $z_i' \sqsubseteq z_i$ (cf. Definition 2.39).

The *state conditional part* of the example requirement Example 6.1 contains two predicates. The predicate "on" reasons about the position of vehicles in regard to a particular neighbor lane. The relation "more than 5 m/s" will evaluate if the velocity of an vehicle relative to the ego vehicle is equal or larger to the defined value (here 5 m/s). In first-order logic, the relation *on* about the vehicle position on a lane can be denoted by a binary predicate $on(x_{pos}, y_{lane}y)$, where both parameters have the type *Lane*; $\alpha(x_{pos}) = \text{Lane}, \alpha(y_{lane}) = \text{Lane}$.

The relation *more than* for the relative velocity can be denoted by a predicate $>(x_{vel}, \text{Higher}), \alpha(x_{vel}) = \text{RelativeVelocity}$. The predicate $>$ takes the abstract value for the current relative velocity of a vehicle as first parameter x_{vel} and compares it to the abstract constant Higher. The abstract constant Higher encapsulates all values for the relative velocity which are larger as 5 m/s. In case of the lane change assistant, a check for equivalence for the relative velocity is sufficient. Efficient evaluations of predicates require types to define an order relation for their abstract values (cf. Section 6.2.2.2.1.

The action part of Example 6.1 is also transformed into a predicate. The predicate $LC(x_{Lane})$ (short for lane change) describes the possibility of a lane change to a given lane. The concrete target lane for the lane change assistant is denoted by the variable $x_{Lane} \in V$ for the example requirement (cf. Eq. (6.23)). For the Example 6.1, The predicate $LC(\text{Left})$ has to be evaluated with the left lane Left as target. Lane changes cannot be expressed by a function because valid statements in first-order logic have to include at least one predicate (cf. Section 2.4.3).

As shown in Fig. 6.9, the predicates of *action part* and the *state condition part* are considered in the definition of the *abstract function* f_A as part of the formulas which represent the overall structure of the requirements. The *abstract function* f_A evaluates objects and their properties based on the predicates from *state condition part* and determines correct and safe actions for the autonomous vehicle system based on the predicates of the *action part*. The *condition part* of the pattern (cf. Fig. 6.4) is considered in the formulas of typed first-order logic to represent the inherent implication between *state condition part* and *action part*. Formulas of typed first-order logic are described in the following section.

6.2.2.2.6. Formulas

A formula consists of at least one predicate with an arbitrary number of terms (cf. Definition 2.39). A formula with one single predicate is also called *atomic formula*. The evaluation result of any formula resides within $\{\text{true}, \text{false}\} \in Boolean$.

Requirements may include multiple predicates (cf. example requirement Example 6.1). Requirements with multiple conditions require the conjunction of atomic formulas in order to correctly replicate such requirements in typed first-order logic. first-order logic defines the logical operators $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \phi \rightarrow \psi$ with ϕ, ψ being formulas. In the context of the runtime monitoring, the operator $\wedge$ is primarily applied for the conjunction of atomic formulas as *state conditional parts*. Conditions which are combined by $\vee$ can be separated into individual requirements. These requirements are evaluated independently from each other. The logical operator $\rightarrow$ is primarily used for the representation of implications between *state conditions* and *system actions* in requirements (cf. Section 6.2.1).

For the runtime monitoring, requirements are expressed by formulas in first-order logic. Formulas maintain the high level structure "state condition $\rightarrow$ action" of requirements — the relationship between *state condition* and *system action* (cf. Section 6.2.1) — which is essential for the evaluation of correct and safe system behavior by the *abstract function* f_A (cf. Fig. 6.9). Beside predicates, the remaining categories of the requirements pattern — *restriction*, *condition*, and *conjunction* — are considered for the definition of formulas which are evaluated by the *abstract function* f_A (cf. Fig. 6.9).

Besides logical operators, first-order logic defines quantifiers $\forall x$ and $\exists x$ in order to quantify over variables (cf. Section 6.2.2.2.3). Quantifiers and variables enable the evaluation of predicates and requirements for a set of objects of a particular type. The Example 6.1 requires that the lane position and relative velocity be evaluated for all vehicles and lanes in the vicinity of the system. In first-order logic, a quantification $\forall x$ over all lanes $\alpha(x) = Lane$ and a quantification $\exists y$ over all vehicles $\alpha(y) = Vehicle$ are defined for the Example 6.1. The predicates within Example 6.1 are evaluated for each combination of vehicle and lane. A lane change must not be performed to the corresponding lane if a vehicle violates one of the given predicates.

Equation (6.23) displays the complete formula of the Example 6.1 in typed first-order logic. The index of objects, functions, and variables (e.g., x_{Lane}, $y_{Vehicle}$, $EGO_{Vehicle}$, pos_{Lane}, $rVel_{\mathbb{R}}$) denotes their types.

$$\forall x_{Lane}(\,\exists y_{Vehicle}(\,On\left(pos_{Lane}\left(y_{Vehicle}\right),x_{Lane}\right)\wedge \\ >\left(rVel_{RelativeVelocity}\left(y_{Vehicle},EGO_{Vehicle}\right),\mathrm{Higher}_{RelativeVelocity}\right)) \\ \rightarrow\neg LC\left(x_{Lane}\right)) \tag{6.23}$$

Types, variables, functions, predicates, and formulas which have been identified in the pattern-based analysis represent the syntactic elements of the typed first-order logic for the runtime monitoring of autonomous vehicle systems. For the semantic interpretation, each element of the typed first-order logic has to be evaluated based on a model $\mathcal{M}$. Such model defines the domain of actual objects, their typing for the interpretation of variable assignments, function symbols and predicate symbols (cf. Section 2.4.4). In the next section, the semantic interpretation of the presented typed first-order logic by the implementation of the runtime monitoring functions (cf. Eqs. (6.1) to (6.4)) is described.

6.2.2.3. Semantic Interpretation of Logic by Implementation

Section 6.2.2.3 has presented the syntactic of the typed first-order logic for the runtime monitoring of autonomous vehicle systems. The syntactic elements — types, variables, function symbols, predicate symbols, and formulas — hold no semantic interpretation. They have to be interpreted about a domain of (real world) objects by evaluating the variables, function, and predicates symbols in formulas, e.g., Eq. (6.23), are substitute by matching domain objects (cf. Section 2.4.4.

For semantic interpretation of the typed first-order logic a model $\mathcal{M}$ has to be defined (cf. Section 2.4.4). The model $\mathcal{M}=\langle\mathbb{D},\delta,\mathcal{I}\rangle$ encapsulates the domain $\mathbb{D}$ of real world objects, a typing function δ, and the interpretation of functions and predicates $\mathcal{I}$. For the runtime monitoring framework, its implementation of types, functions, and predicates represents the semantic interpretation of of the typed first-order logic.

6.2.2.3.1. Domain

In the real world, the domain of possible objects $\mathbb{D}_{\text{real}}$ is infinite even though the set of types which has been derived from requirements is finite. Following the open-world assumption (cf. Section 3.8.1), the real-world as the domain of autonomous vehicle systems has infinite many arbitrary objects or finite but continuously increasing object sets for some types. The set of vehicles in the real world is a finite domain but increases continuously while the number of real traffic situations is infinite. Furthermore, the real word is subject to continuous change because many real-world objects, e.g., vehicles or pedestrians, operate autonomously.

The runtime monitoring evaluates autonomous vehicle systems and their environments at concrete time stamps $t_i\in\mathbf{T}$ — $\mathbf{T}$ denotes the time domain — based on runtime data from the preprocessing of the autonomous vehicle systems about the *system state* (cf. Fig. 5.1). The *system state* encapsulates the internal state of the autonomous vehicle

system and the state of its environment (cf. Definition 3.6). The time stamps of the runtime monitoring commonly correspond to the processing cycles of the autonomous vehicle systems. Autonomous vehicle systems sense their environment and process corresponding decisions in predefined cycles, e.g., all 10 ms.
The elements in the domain $\mathbb{D}$ for the *system state* are finite in each processing cycle. Even though the number of objects in the real world is theoretically infinite only a finite number of the number of objects with fixed types can be present in the system's environment for each processing cycle. The values of the internal system parameters are also known and fixed. It yields $\mathbb{D} \subset \mathbb{D}_{\text{real}}$. As a result, the runtime monitoring always evaluates a known and finite number of objects and parameters for the *system state*.
As the real world continuously changes over time, the number of objects in the system's environment and the values of the internal system parameter change between two processing cycles of the autonomous vehicle systems. For example, vehicles enter or leave the perception range of the automated ego vehicles' sensors. The domain $\mathbb{D}$ of the *system state* for two subsequent processing cycles t_1 and t_2 need not to be identical.
Over time, a *trace* of *system states* emerges (cf. Definition 2.32). For the autonomous vehicle systems, system states are formed by the values of all internal system parameters and the positions and behaviors of objects in the systems' environments as a representative within domain $\mathcal{D}_S^I$. The runtime monitoring also encounters traces of *abstract situations* as abstract representations of these system states within domain $\mathcal{D}_A^I$.
A *trace* of *system states* is defined by $\overline{\mathbb{D}} = \mathbb{D}_1\mathbb{D}_2, \ldots$ with $\mathbb{D}_i$ denoting the system state at time stamp t_i. All timestamps $t_1 < t_2 < \ldots$ are timely ordered. At a time stamp only one domain $\mathbb{D}$ can exist but any domain $\mathbb{D}$ can be present at multiple time stamps t_i.
While domains $\mathcal{D}_S^I$ and $\mathcal{D}_S^O$ (cf. Fig. 6.2) are realized in the development of autonomous vehicles systems and are provided for the implementation of the runtime monitoring framework, domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$ (cf. Fig. 6.2) are subject to the development of the runtime monitoring framework. The implementation of types from $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$ in object-oriented programming languages by the runtime monitoring framework inherently provides their semantic interpretation.
The implicit typing of object-oriented programming languages is beneficial for the typing of objects by the typing function δ. The typing function δ is described in the following section.

6.2.2.3.2. Typing Function

The typing function $\delta : \mathbb{D} \to \mathcal{T}_D$ of the model $\mathcal{M}$ is implicitly implemented by the explicit typing of object-oriented programming languages, e.g., C++ and Java. As the types of input data for the runtime monitoring framework — the data elements of domains $\mathcal{D}_S^I$ and $\mathcal{D}_S^O$ — are given by the implementation of the autonomous vehicle systems, the typing of data elements for domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$ results from the application of the runtime monitoring functions f_I and f_O to this input data.
In case no special type has yet been defined for a domain object because it has not yet been known, the typing function δ assigns the empty type $\perp$ to this object. In programming languages, e.g., C++ and Java, these objects are denoted by *null*. However

these unknown objects should have been detected as failure by the perception of the autonomous vehicle systems (cf. *preprocessing* in Fig. 5.1).
The distinction of dynamic and abstract typing (cf. Section 6.2.2.2.2) has a minor relevance for the implementation of the runtime monitoring framework. Dynamic and abstract types have to be identically considered in the data structure of domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$. Nevertheless, objects as resulting from the *input abstraction* f_I and *output abstraction* f_O can only be of dynamic types and never of abstract types (cf. Section 2.4.1). Abstract types are not defined by requirements but are manually added in order to be used in the implementation of functions, predicates, and formulas, for optimizations (cf. Section 6.2.2.2.2).

6.2.2.3.3. Interpretation of Functions and Predicates

The interpretation function $\mathcal{I}$ of the model $\mathcal{M}$ assigns particular functions or predicates to the function symbols and predicate symbols of the typed first-order logic. Methods represent these particular functions or predicates in the software implementation of the runtime monitoring framework. The signature of these methods matches the signature — identical number and typing of parameters — of corresponding function symbols and predicate symbols of the typed first-order logic. While the numbers and types of concrete objects of domains $\mathcal{D}_S^I$, $\mathcal{D}_S^O$, $\mathcal{D}_A^I$, and $\mathcal{D}_A^O$ are know at runtime, the interpretation of functions and predicates can be defined at design time.
The function symbols of the typed first-order logic are mapped to properties of types from domain $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$. The implementations of *input abstraction* f_I and *output abstraction* f_O process the property values for types in the domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$ based on the objects and their properties in the domains $\mathcal{D}_S^I$ and $\mathcal{D}_S^O$. The processed function values are saved as abstract values in struct variables resp. class attributes for the corresponding objects in domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$. The types of struct variables resp. class attributes match the return types of corresponding function symbols.
In the evaluations of formulas by the *abstract function*, the abstract values of properties can directly be accessed via the struct variables resp. class attributes. The method implementations of the functions do not have to be revised. Get-methods can be introduced for the access of struct variables resp. class attributes in order to maintain the operational aspect of functions in the evaluation by the *abstract function* f_A.
Predicates are not part of the abstraction f_I, and f_O but are evaluated by the *abstract function* f_A. Therefore, predicates are implemented as methods with identical signature as the corresponding predicate symbols of the typed first-order logic. In general, the value for this predicate is processed by its implementation each time the *abstract function* requires the value of this predicate for the evaluation of formulas. Nevertheless, results for each predicate in conjunction with the passed objects can be saved in variables in order to avoid repeated processing of identical system data. The variables have to match the return type of the corresponding predicate symbols.
The implementation of functions f_I, f_O, and f_A of the runtime monitoring framework is described in more detail in the following section.

6.2.3. Implementation of Runtime Monitors

The evaluation of autonomous vehicle systems in simulations and during operation requires the implementation of *runtime monitors*. Runtime monitors (cf. Section 2.3.1) implement the domain, variables, functions, predicates, formulas of the system requirements and evaluate them on available data from the autonomous vehicle systems.

In comparison to the terminology of [LS09], the runtime monitoring framework depicts a monitoring approach which monitors conditions *online* and *offline*. The runtime monitoring can be used *online* — during operation of the autonomous vehicle system — for the enforcement of the system's safety (cf. Section 7.1.1). *Offline*, engineers analyze recorded traces of *abstract situations* for the improvement of the autonomous vehicle system and the definition of new test cases.

Runtime monitors can be implemented in software, hardware or in both as hybrid monitors (cf. Section 2.3.1). The implementation of runtime monitors as software has the advantage that the software monitors can be used in throughout all activities in the development of autonomous vehicle systems without the costly assembly and integration of hardware components, e.g., FPGAs (cf. Section 3.1.2). This advantage is especially relevant for software-based simulations in the system verification (cf. Section 3.6.2). For the lane change assistant, the runtime monitoring framework is implemented in the object-oriented programming language C/C++.

Runtime monitors have to access the runtime data of the autonomous vehicle system via defined interfaces for the runtime monitoring. These interfaces by the system have to provide sufficient data about the inputs and outputs of the monitored system *function* (cf. Section 6.2.1). The following section discusses the implementation of the data access for the runtime monitoring of autonomous vehicle systems.

6.2.3.1. Implementation of Data Access

Car manufacturers do not widely recognize the benefits of additional runtime monitoring capabilities. Software and hardware of embedded systems in the automotive domain are highly optimized (cf. Section 2.3). ECUs currently offer only limited to none capacity for additional data processing other than planned vehicle's control software. Additional software on these ECUs would have an immediate safety-critical impact on the processing of the autonomous vehicle system. The implementation of the runtime monitoring framework has to cope with the limited resources of these embedded systems as long as no additional data storage, and processing capabilities are provided for the runtime monitoring.

The autonomous vehicle system in the simulations and during operation in the real world have to be identically. Otherwise, results from the runtime monitoring of the autonomous vehicle system in simulations of the system verification are not applicable to the operation of autonomous vehicle systems in the real world. The exclusive usage of specialized hardware and code instrumentation of the autonomous vehicle system for the runtime monitoring in the system verification is not possible. The data access, processing

— including the timing behavior — of the runtime monitoring must not interfere with the processing of the autonomous vehicle systems.

As shown by Definition 2.32, the autonomous vehicle system, and its environment are evaluated by the runtime monitoring framework at concrete timestamps. The content of domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$ has to sufficiently match the state of the autonomous vehicle system and the state of its environment at each time stamp. Otherwise, the results from the runtime monitoring framework are not sound. The input data from the autonomous vehicle system — domains $\mathcal{D}_S^I$ and $\mathcal{D}_S^O$ — are updated in defined progressing cycles. The processing of the runtime monitoring framework has to be aligned with the processing of autonomous vehicle systems. For many autonomous vehicle systems, the processing cycle of the runtime monitoring framework matches the processing cycle of the autonomous vehicle system.

The processing cycle of the runtime monitoring framework can be a multiple of the autonomous vehicle system in order to reduce the computational requirements of the runtime monitoring. The processing of the runtime monitoring has to be integrated into the scheduling of the autonomous vehicle system without any side effects to the processing of the autonomous vehicle system. In cases, the processing of the runtime monitoring framework must neither interfere with processing of the autonomous vehicle system nor miss safety-critical behavior of the autonomous vehicle system.

In case an autonomous vehicle system has multiple processing cycles, e.g., the updates of the system behavior due to different cycles of sensor information, the greatest common divisor for all system cycles could be used. However, this approach has the disadvantage of unnecessary processing on already evaluated system data. Another approach is to have multiple cycles for the runtime monitoring with each cycle being aligned to one system processing cycle. This approach avoids any unnecessary processing but might introduce more complexity in the scheduling of the runtime monitoring within the autonomous vehicle system.

The implementation of the *qualitative runtime monitoring* separates itself in the definition of domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$ and the implementation of functions f_I, f_A, and f_O (cf. Fig. 6.2). For the *quantitative runtime monitoring*, the recording and comparison of *abstract situations* from the *input abstraction* f_I has to realized.

6.2.3.2. Implementation of Runtime Monitoring Domains

The explicit typing of object-oriented programming languages, e.g., C++ and Java, is beneficial for the implementation of domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$. In the implementation of these domains, two types have to be differentiated; types are describing complex real-world objects and types which describe ranges of possible values.

Types describing complex real-world objects can be directly implemented as structs or classes in object-oriented programming languages, e.g., C++ or Java. The inheritance between classes allows to identically rebuild the type hierarchy of these domains in the software. Structs or classes allow encapsulating properties of objects as struct variables resp. class attributes.

Types defining abstract value ranges, e.g., the position of vehicle relative to the ego vehicle (*before*,*next*, *behind*), are implemented as enumerations. Each concrete object defined for these types in the typed first-order logic is implemented as a corresponding element in the enumeration and extends by the element *OOR* for out-of-range values.
For the domains $\mathcal{D}_A^I$ and $\mathcal{D}_A^O$ an additional struct resp. class is added in order to store the current state and objects for each domain. For this *container* class, the set of current objects $\mathbb{D}^z$ for each type is saved in an individual struct variable resp. class attribute (cf. Section 2.4.4). These struct variables resp. class attributes allow the evaluation of variables in formulas by iterating over the set of objects of the specific type in the corresponding struct variable resp. class attribute. For the lane change assistant, one single domain has been implemented because the output domain is a subset of the input domain.
For the runtime monitoring framework, the functions f_I, f_A, and f_O of the *qualitative runtime monitoring* have to be implemented (cf. Fig. 6.2). The input and *output abstraction* uses functions of the typed first-order logic to process the properties of abstract types. Predicates of the typed first-order logic are used by the *abstract function* to process correct and safe system actions. The implementation of the *input abstraction* f_I and *output abstraction* f_O is described Section 6.2.3.3 while the implementation of the *qualitative runtime monitoring* with the *abstract function* is presented in Section 6.2.3.4. Additional to the qualitative runtime monitoring, the *quantitative runtime monitoring* is described in Section 6.2.3.5.

6.2.3.3. Transformations between Domains

The runtime monitoring framework depicts the *function* as a *black-box* and requires the autonomous vehicle systems to provide the domains $\mathcal{D}_S^I$ and $\mathcal{D}_S^I$ as interfaces for the access of the *function's* inputs reps. outputs. The *input abstraction* f_I and *output abstraction* f_O have to process the current abstract representation of the current system and environment state from the currently available data at the interfaces of the autonomous vehicle systems. Data objects forming the domains $\mathcal{D}_S^I$ resp. $\mathcal{D}_S^O$ of the autonomous vehicle systems have to be transformed into objects of domains $\mathcal{D}_A^I$ resp. $\mathcal{D}_A^O$ for the runtime monitoring framework. The provision of system data for the *input abstraction* f_I and *output abstraction* f_O by the autonomous vehicle systems via the interfaces will not require any instrumentation of source code (cf Section 2.3.1) if these interfaces haven been considered in the design and implementation of the autonomous vehicle systems.
The *input abstraction* f_I and *output abstraction* f_O represent model-to-model transformation (cf. [CH03]) between the corresponding domains $\mathcal{D}_S^I$ and $\mathcal{D}_A^I$ resp. $\mathcal{D}_S^O$ and $\mathcal{D}_A^O$. For *input abstraction* f_I and *output abstraction* f_O a set of transformation rules are defined. These transformation rules define how objects of domains $\mathcal{D}_A^I$ resp. $\mathcal{D}_A^O$ are processed from the current objects available in domains $\mathcal{D}_S^I$ and $\mathcal{D}_S^O$ of the autonomous vehicle system. Figure 6.10 displays the transformation of the *input abstraction* f_I between domains $\mathcal{D}_S^I$ and $\mathcal{D}_A^I$. The objects of domains $\mathcal{D}_S^I$ and $\mathcal{D}_A^I$ are the only elements of the *input abstraction* f_I which are solely known at runtime. All other elements — types, transformation, and transformation rules — of the abstraction are defined at development

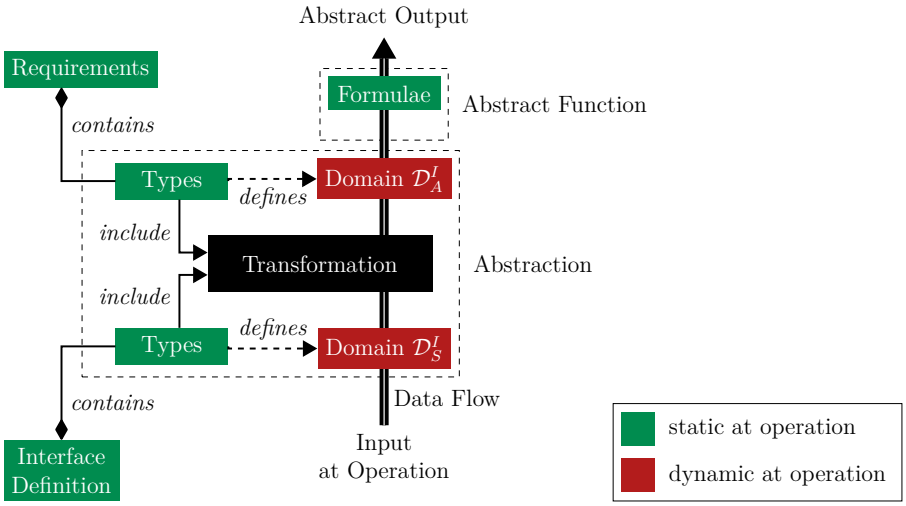


Figure 6.10.: Transformation between domains $\mathcal{D}_S^I$ and $\mathcal{D}_A^I$.

time because they do not change at runtime. The same applies to the *output abstraction* f_O.

Transformation rules commonly consist of two parts; a left-hand side (LHS) and a right-hand side (RHS) (cf. [CH03]). The LHS accesses the input model (here, domains $\mathcal{D}_S^I$ or $\mathcal{D}_S^O$) and the RHS interferes with the target model (here, domains $\mathcal{D}_A^I$ or $\mathcal{D}_A^O$) (cf. [CH03]). For the LHS, the numbers and types of objects from the input model are defined for the transformation. The RHS defines the numbers and types of objects in the target model which are created or adapted by the transformation rule. Transformation rules define the mapping and calculation between properties of objects from the LHS to properties of objects in the RHS. Property values of objects in the RHS are calculated based on the values of a set of properties from the objects of the LHS. For example, the relative velocity of a vehicle for the domain $\mathcal{D}_A^I$ is calculated based on the velocities of this vehicle and the automated ego vehicle in the domain $\mathcal{D}_S^I$.

The calculations of object properties for the target model (RHS) match the functions of the typed first-order logic for the corresponding object type. Unlike the typed first-order logic, the calculations of object properties in the *input abstraction* f_I and *output abstraction* f_O take objects from the input domains $\mathcal{D}_S^I$ resp. $\mathcal{D}_S^O$ as inputs and not from the target domains $\mathcal{D}_A^I$ or $\mathcal{D}_A^O$. The results of transformation rule for object properties match the envisaged results of functions symbols of the first-order logic.

For the lane change assistant, the *input abstraction* f_I transforms vehicles from the domain $\mathcal{D}_S^I$ to the domain $\mathcal{D}_A^I$. For an object of the type *vehicle* in the domain $\mathcal{D}_S^I$, a transformation rule creates a object of type *vehicle* in the domain $\mathcal{D}_A^I$. Additional to basic information about vehicle objects, e.g., identifiers, the transformation rule processes properties *lane position*, *relative position*, and *relative velocity* for type *vehicle*:

Lane position The property *lane position* of a vehicle object in the domain $\mathcal{D}_A^I$ is calculated by comparing the real world position ($\mathbb{R}$) of the vehicle with the dimensions of lanes in domain $\mathcal{D}_S^I$. The result of this transformation is LanePosition $\in$

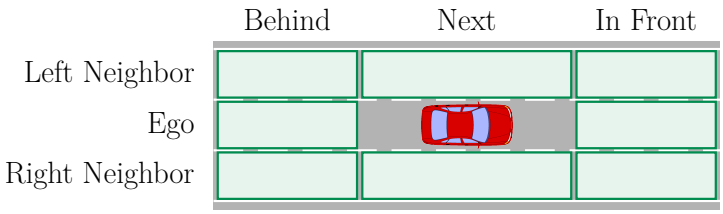


Figure 6.11.: Zoning as representation of vehicle positions in domain $\mathcal{D}_A^I$.

$\{Left, Ego, Right\}$ where *Ego* denotes the lane the automated vehicle is currently driving and *Left* resp. *Right* represent the left resp. right neighbor lane.

Relative Position The relative position of a vehicle object in the domain $\mathcal{D}_A^I$ is calculated by comparing the vehicle's position in driving direction with the position and dimension — front and rear end (cf. Fig. 3.8) — of the automated vehicle. The *relative position* in domain $\mathcal{D}_A^I$ is categorized into *behind* the automated vehicle, *next* to the automated vehicle, and *before* the automated vehicle; RelPosition $\in$ $\{Next, Before, Behind\}$.

Relative Velocity The property *relative velocity* of a vehicle object is calculated based on the velocity difference between this vehicle and the automated vehicle. For the abstract representation in domain $\mathcal{D}_A^I$, the *relative velocity* is categorized in regard to the predefined criteria reps. limits of predicates about the relative velocity for the lane change assistant, the type for the *relative velocity* has the domain RelVelocity*in* $\{LOWER, HIGHER, OOR\}$ (cf. Section 6.2.2.2.1)

For the transformation of vehicle objects, the LHS of the transformation rule takes objects of types *Vehicle* and *Lane* from domain $\mathcal{D}_S^I$ — precisely the examined vehicle, the automated vehicle, and the driving lane of the examined vehicle. As result of the transformation, the RHS creates an abstract object of type *Vehicle* and it properties in domain $\mathcal{D}_A^I$. The data structure of the *abstract situation* ($\mathcal{D}_A^I$) in the context of the lane change assistant is described in the appendix by Fig. 7.7.
For the lane change assistant, the position of vehicles in the the domain $\mathcal{D}_A^I$ by its *lane position* and *relative position* can depict as *zones*. As shown by Fig. 6.11, seven zones are located around the automated ego vehicle with the zone in which the automated vehicle (red vehicle) is located is being omitted. No other vehicle can drive within the zone of the automated ego vehicle unless a collision has occurred. In driving direction of the automated vehicle, the zones refer to — from front to back — *before*, *next*, *behind*. The zones in lateral direction to the automated vehicle relate to the driving lane of vehicle — *left*, *ego*, *right*. In the front and back the extension of the zones is limited by the sensor perception range, while the zones in the lateral direction are limited by the size of the three considered lanes. All zones are relative to the automated ego vehicle and change with the behavior of the automated ego vehicle. For example, the lanes are moved in the direction of any lane change by the automated ego vehicle. A special case are lanes which do not exist because, e.g., the highway only has two lanes or the automated vehicle

drives on the leftmost lane of a highway. The type of such lanes is *non-existing* and the zones for these lanes are omitted (cf. Section 3.3.1.2).
Each abstraction of the runtime monitoring framework is defined by a set of transformation rules. For the runtime monitoring framework, the transformation rules within *input abstraction* f_I and *output abstraction* f_O are unidirectional — rules only process from the domains of the autonomous vehicle system towards the domains of the runtime monitoring framework but not the other way around. The set of transformation rules is sequentially applied at all objects of the abstraction's input domain $\mathcal{D}_{In}$. The input domain $\mathcal{D}_{In}$ corresponds to the domains of the autonomous vehicle system $\mathcal{D}_S^I$ for the *input abstraction* resp. $\mathcal{D}_S^O$ and for the *output abstraction*. Algorithm 1 displays the algorithm for abstractions between complex domains of *input abstraction* f_I and *output abstraction* f_O. Each processing cycle, the algorithm commences by initializing the container object for the model of output domain $\mathcal{D}_{Out}$. The output domain $\mathcal{D}_{Out}$ corresponds to the $\mathcal{D}_A^I$ for the *input abstraction* resp. $\mathcal{D}_A^O$ and for the *output abstraction*. The Initialization of the container object includes creating a new object or resetting the existing container object.

Algorithm 1: Basic implementation of input abstraction f_I and output abstraction f_O.

Data: Instance d_{In} of input domain $\mathcal{D}_{In}$
Result: Instance d_{Out} of output domain $\mathcal{D}_{Out}$
1 initialize d_{Out};
2 **foreach** *Object* o_{In} *of type* t_{In} *in* $\mathcal{D}_{In}$ **do**
3 | **foreach** *Rule* $R(\dots)$ *applicable for type* t_{In} **do**
4 | | $\mathcal{O}_{Out} \leftarrow R(o_{In})$;
5 | | $d_{Out} \leftarrow \mathcal{O}_{Out}$;
6 | **end**
7 **end**

The algorithm iterates over the objects $\mathcal{O}_{In}$ of the input domain $\mathcal{D}_{In}$ (Line 2). For each object $o_{In} \in O_{In}$ of type t_{In}, the set of applicable transformation rules is determined. Each rule is applied to the object o_{in} (Line 3). The transformation rule creates new objects $\mathcal{O}_{Out}$ of types t_{Out} for the output domain $\mathcal{D}_{Out}$ which are related to type t_{In} by the transformation rule. Subsequently, the transformation rule calculates properties for the create objects $\mathcal{O}_{Out}$ (Line 4). Even though, rules are selected for one domain object o_{In}, transformation rules may incorporate multiple objects from the input domain $\mathcal{D}_{In}$ in the processing of properties for output objects $\mathcal{O}_{Out}$.
The new objects $\mathcal{O}_{Out}$ are added to the model of the output domain $\mathcal{D}_{Out}$ (Line 5). After all rules for one object o_{In} have been processed, the algorithm continues with the remaining objects of the input domain $\mathcal{D}_{In}$ until all of the objects have been processed. For the correctness and soundness of *input abstraction* f_I and *output abstraction* f_O, all transformation rules have to be independent from each other and only create new objects for the output domain $\mathcal{D}_{Out}$. Otherwise, existing objects may be altered or updated

by multiple transformation rules leading to a potential contradictory transformation of domain objects. For transformations which update identical domain objects, the order of execution is important for these rules. Otherwise, transformation rules may be executed which require objects for updating which have not yet been created by another transformation rule.

The algorithm Algorithm 1 is nondeterministic. For the same set of input objects of the domain $\mathcal{D}_{In}$ the algorithm processes the same output model for domain $\mathcal{D}_{Out}$. However, the order of transformation rules which are applied to an object o_{In} is arbitrary. Intermediate states of the output models may vary even though the final output model is identical. The termination of algorithm Algorithm 1 is subject to the maximum number of objects possible for the input domain $\mathcal{D}_{In}$. Algorithm Algorithm 1 may not terminate if the maximum number of objects for any type of input domain $\mathcal{D}_{In}$ is infinite.

In the worst case, the algorithm Algorithm 1 has a complexity of $\mathcal{O}(n^2)$ — where n denotes the number of object for the input domain $\mathcal{D}_{In}$. The algorithm iterates over the n objects of the input domain $\mathcal{D}_{In}$ and applies all transformation rules to each element. Each of these transformation rules may also iterate over the complete set of n input objects from $\mathcal{D}_{In}$ in order to correctly process the new objects and their properties for the output domain $\mathcal{D}_{Out}$.

The resource consumption of Algorithm 1 is subject to the number of objects in domain $\mathcal{D}_{In}$ at runtime. As long as $\mathcal{D}_{In}$ has a upper bound for the number of objects in domain $\mathcal{D}_{In}$ at runtime, the worst case resource consumption of Algorithm 1 can be estimated at design time. Otherwise, the resource consumption of Algorithm 1 has to be monitored at runtime in order to exclude any impact on the processing of the autonomous vehicle system.

For many autonomous vehicle systems, the *function* (cf. Fig. 5.1) solely process as output one single action. For single value function outputs, the *output abstraction* f_O does not require algorithm as complex as Algorithm 1. A single transformation rule is sufficient.

The lane change assistant processes a single target point as output (cf. Fig. 3.13). This target point is abstracted by the *output abstraction* f_O to a *target lane* Target $\in \{Left, Ego, Right\}$ of type *Lane* by comparing the position of the target point to the boundaries of each lane. This simplification reduces the complexity of the *output abstraction* f_O to $\mathcal{O}(1)$. The data structure of the domain $\mathcal{D}_A^O$ in context of the lane change assistant is shown in Fig. A.1.

The technical implementation of the abstractions depends on the deployment of the runtime monitoring framework. In case the framework is deployed on the same ECU as the autonomous vehicle system, shared variables can be used for the storage and access to the system data. A shared variable is created for each data item of the interface between the runtime monitoring framework and the autonomous vehicle system. The autonomous vehicle systems updates these shared variables with the system data of the current processing cycle. The runtime monitoring framework reads the data from these shared variables but does not update the shared variables. The transfer of data from the runtime monitoring framework to the autonomous vehicle system requires the addition of shared variables.

In case the runtime monitoring framework and the autonomous vehicle system are separated on different ECUs, the autonomous vehicle system has to publish the necessary system data for the runtime monitoring on communication networks, e.g., CAN or FlexRay, which connect both ECUs. In case the existing hardware platform (cf. Section 3.3.2.2) is incapable of providing the system data, additional messages have to be added for the communication networks, or additional communication connection have to be added between unconnected ECUs.
As shown in Fig. 6.10 results of the *input abstraction* — the abstract representational of the current state of autonomous vehicle system and its environment — are evaluated by the *abstract function* f_A. The implementation of the *qualitative runtime monitoring* is described in the following section.

6.2.3.4. Implementation Qualitative Monitoring

The implementation of the *qualitative runtime monitoring* (cf. Section 5.4), is concerned with the realization of *abstract function* and *conformity oracle*. For later improvement of the autonomous system, the *qualitative runtime monitoring* has to consider the logging of *abstract situations* with critical and unsafe behavior of autonomous vehicle systems. The implementation of the *abstract function* f_A and *conformity oracle* are described in the following sections. The logging is presented in Section 6.2.3.6,

6.2.3.4.1. Abstract Function

The *abstract function* f_A evaluates the behavior of the autonomous vehicle system with regard to its requirements. While types, domains, and functions are considered in the *input abstraction* and *output abstraction*, predicates and formulas are incorporated in the evaluation of the *abstract function*. Each processing cycle, formulas and predicates are evaluated based on the current *abstract representation* from the *input abstraction* f_I. All predicates of a formula's *state condition part* are individually evaluated based on the current objects in the *abstract representation*. In most cases, the evaluation of predicates comes down to the comparison of objects for equivalence because predicates and their criteria have already been considered in the definition of the domain $\mathcal{D}_A^I$.
The result of formula's *state condition part* yields from the evaluation results of all its predicates. A valid *state condition part* implicates the application of the formula's *action part*. Abstract actions are either included in or excluded from the set of correct and safe actions for the autonomous vehicle system. The set of correct and safe actions for the autonomous vehicle system represents the output and result of the *abstract function*.
The finiteness of the action set for the autonomous vehicle system impacts the implementation of the *abstract function* f_A:

Infinite set of actions: The infinite set of abstract actions for an autonomous vehicle system cannot be comprehensively considered in the implementation of the *abstract function* f_A. As result, a set of required abstract actions and a set of invalid abstract actions have to be considered in the implementation (cf. Algorithm 2).

Finite set of actions: A finite set of abstract actions for an autonomous vehicle system can be used in the implementation of *abstract function* f_A as an initial reference set from which abstract actions are removed based on the evaluation of formulas from the typed first-order logic.

Both implementations of *abstract function* f_A are presented in the following.
An infinite number of abstract system actions for the autonomous vehicle system requires the *abstract functions* f_A to explicitly consider a set for necessary actions and a set for invalid actions. Algorithm 2 displays the implementation for the abstraction function f_A for a infinite set of abstract system actions. Other monitoring algorithms are investigated in [PDE11].

Algorithm 2: Function implementation of the *abstract function* f_A.

Data: Instance d_A of Domain $\mathcal{D}_A^I$
Data: Set F of first-order formulas
Result: Set l of system actions

```
1 initialize set l;
2 foreach Formula f ∈ F with f : a_f ← c_f do
3 |   if state conditional part c_f is valid then
  |   |   /* For the set of required abstract actions, the addition of
  |   |      actions to set l may throw an error                      */
4 |   |   l ← actions A ∈ a_f;
5 |   end
6 end
7 return l;
```

The formulas F of the typed first-order logic (cf. Section 6.2.2.2.6) are separated into two different sets F_{req} and F_{inv} based on their restriction of system actions. The restriction of system actions corresponds to the optional category *negation* of the analysis pattern (cf. Fig. 6.4). F_{inv} encompass all formulas whose *action parts* contain negations — restrictions on the execution of corresponding abstract actions for the *state condition*. F_{req} denotes all formulas which contain no negations in their *action parts*. Formulas in F_{req} require abstract actions to be explicitly performed by the autonomous vehicle system in situations matching the corresponding *state conditions*. In this work, it is assumed that all formulas contain sets of solely negated actions or solely unnegated actions in order to evaluate each requirements in specific functions without any adaptations. Formulas containing both negated actions and unnegated actions are excluded. It yields $F_{req} \cup F_{inv} = F \wedge F_{req} \cap F_{inv} = \emptyset$.
The implementation of the *abstract function* f_A for autonomous vehicle systems with infinite set of abstract actions is separated into two functions. These functions process the set of required abstract actions and the set of invalid *abstract function* independently of each other by evaluating the corresponding set of formulas F_{req} and F_{inv}. However, the implementation of both functions is similar and primarily differentiates in the sets

F_{req} and F_{inv}. Algorithm 2 represents the template for both function implementations with $F = F_{req}$ resp. $F = F_{inv}$.
Both functions receive the current state representation from the *input abstraction* f_I and formula set F_{req} resp. F_{inv} as arguments. The algorithm iterates over all formulas of these sets (Line 2) and evaluates their *state condition parts* c_f (Line 3). In case the condition part is valid, the actions from the *action part* of the formulas is added to set l. Set l represents the output of the function. Results from both functions — sets l_{req} and l_{inv} — are forwarded to the *conformity oracle* in order to evaluate the correctness and safety of the real action by the autonomous vehicle system (cf Section 6.2.3.4.2).
In case the *condition parts* of formulas from F_{inv} have been evaluated as valid (*true*), the invalid abstract actions in the *action parts* $A \in a_f$ can be directly added to the result set l_{inv} (cf. l in Line 4). Invalid abstract actions represent alternatives because each action describes a reduction of the space in which the autonomous vehicle may safely operate. Therefore, the connection of invalid abstract actions by conjunction ($\wedge$) or disjunctions ($\vee$) has no impact on the result set l_{inv} for the *abstract function* f_A.
In comparison to invalid abstract actions, the addition of mandatory abstract actions to set l_{req} of mandatory actions (cf. l in Line 4) is restricted and may lead to irrational results. In an *abstract situation*, requirements may not require more than one abstract action to be performed. Required abstract actions in an *abstract situation* are only allowed to be connected by a disjunction ($\vee$) but never by a conjunction ($\wedge$). Requirements may define alternatives of abstract actions in their *action parts* of whom one *abstract action* must be performed by the function.
When adding a set of alternative abstract actions l_{add} to the set l_{req}, the intersection of set l_{req} and l_{add} is stored as new set l^*_{req} unless set l_{req} is empty; $l^*_{req} = l_{req} \cap l_{add}$. An empty result set l^*_{req} indicates inconsistencies in the formulas of the typed first-order logic and in the corresponding requirements. For example, formulas for the lane change assistant require two different sets of abstract actions, e.g., $l^1_{req} = \{LC_{\text{left}}\}$ and $l^2_{req} = \{LC_{\text{right}}\}$ in one single *abstract situation*. The intersection $l^1_{req} \cap and l^1_{req}$ for the set of required action results in an empty set l^*_{req} because no abstract action is contained in both sets. No unique action can be identified for the system in this *abstract situation*.
A finite set of all abstract actions can be used as the initial reference set for the evaluation of correct and safe system behavior by the *abstract functions* f_A. Invalid actions are removed by *abstract function* f_A from the reference set of all abstract actions until all formulas of the typed first-order logic have been evaluated and the set of safe and correct actions remains. The lane change assistant is a concrete example of an autonomous vehicle system with a finite set of system actions. Neglecting the parametrization by velocity and type of the target point (cf. Fig. 3.13), the lane change assistant can either decide to remain on its current lane or to change to the left resp. right neighbor lane.
As displayed by Algorithm 3, the set of abstract actions for the lane change assistant is used in the implementation of the *abstract function* f_A as initial reference set (Line 1). Formulas of the typed first-order logic are successively evaluated by the *abstract function* f_A. In case the state conditions part of a formula is valid (*true*) for the current *abstract situation* (Line 3), the actions of the formula's *action part* are removed from the reference set (Line 4). The consideration of required actions is not necessary for the lane change

assistant because all requirements for the lane change assistant only define restrictions on the execution of lane changes. There exist no requirement for the lane change assistant which requires the execution of actions for a specific (abstract) situation. As result of the *abstract function* f_A for the lane change assistant, the set of correct and safe actions remains.

Algorithm 3: Implementation of *abstract function* f_A for the lane change assistant.

Data: Instance d_A of Domain $\mathcal{D}_A^I$
Result: Set l_O of correct and safe system actions

```
1 initialize l_O with the complete set of possible actions;
2 foreach Formula F : a_F ← c_F do
3 |   if state conditional part c_F is valid then
4 |   |   remove action a_F from l_O;
5 |   end
6 end
7 return l_O;
```

In case a formula of the typed first-order logic requires the autonomous vehicle system to execute a specific action, all other actions have to be removed from the reference set of abstract actions. The require *abstract action* is the only action that remains as the result of the *abstract function* f_A. The soundness of *abstract function* f_A requires the typed first-order logic to contain no contradicting formulas. Otherwise, multiple actions could be required for a single *abstract situation* or an abstract action for a single *abstract situation* could be required by formula while another formula evaluates the action as invalid.

The time and space complexity of the *abstract function* f_A is subject to the size of the formula set and the number of predicates contained by these formulas. The iteration over predicates is not explicitly displayed in Algorithm 2 and in Algorithm 3 but has to be included for the evolution of the *state conditional parts*. In the worst case, the *abstract function* has a complexity of $\mathcal{O}(n \cdot m)$ where n denotes the number of formulas and m the maximum number of predicates of these formulas.

In the worst case, the memory consumption of *abstraction function* f_A requires one data object for each predicate in each formula, for each formula result, and each system action in the result set l.

In case the input history is not considered by the *abstract function* f_A, Algorithm 2 is determined but not deterministic. For the same input the *abstract function* f_A provides the same result because the output solely depends on the input situation. The order in which formulas and predicates are evaluated is not defined and may change leading to varying internal states of result sets l_{req} and l_{inv} for repeated evaluations. In case the input history is considered by the *abstract function* f_A, Algorithm 2 is nondeterministic. The result sets l_{req} and l_{inv} are subject to the history of input *abstract situations* which is difficult if not impossible to be fully defined for the *abstract function* f_A.

The result from the *abstract function* f_A are compared by the *conformity oracle* with the actual abstract action processed by the autonomous vehicle systems (cf. Section 5.4.2). The implementation of the *conformity oracle* is described in the following section.

6.2.3.4.2. Conformity Oracle

The *conformity oracle* takes the sets of required and invalid system actions from the *abstract function* f_A and the abstracted output action from the *output abstraction* f_O as input. Each processing cycle, the oracle evaluates the correctness and safety of the autonomous vehicle system by evaluating if the action from the autonomous vehicle system meets the abstract actions from the *abstract function* f_A.
The implementation of the *conformity oracle* is impact by the finiteness of abstract action set for the autonomous vehicle system in domain $\mathcal{D}_A^O$ (cf. Section 6.2.3.4.1). In case the *abstract function* f_A provides the *conformity oracle* with two sets — the set of required actions l_{req} and the set of invalid actions l_{inv} — (cf. Section 6.2.3.4), the *conformity oracle* compares the abstract system action from the *output abstraction* f_O with both sets l_{req} and l_{inv}. The behavior of the autonomous vehicle system is evaluated as correct and safe by the *conformity oracle* if

- the set l_{req} of required actions from the *output abstraction* f_O is not empty and the abstract action from the *output abstraction* f_O is included in l_{req}, or
- the set l_{req} is empty and the abstract action from the *output abstraction* f_O is not included in the set l_{inv} of invalid actions from the *output abstraction* f_O.

In any other case, the correctness and safety of the system behavior cannot be guaranteed. Besides the evaluation by *in-set* relations, the comparison by the *conformity oracle* can be implemented as probability-based classification (cf. [Agg14; MPB09]).
The *qualitative runtime monitoring* must be able to identify contradicting statements in formulas of the typed first-order logic in order to ensure the validity of its results. The complexity of the real world may make it difficult — if not impossible — to identify all contradicting statements in requirements at design time completely. In every processing cycle, the *conformity oracle* has to cross-check items of sets l_{req} and l_{inv} with each other in order to ensure that no action is included in both sets. A necessary but invalid system action would indicate an inconsistency in the requirements of the autonomous vehicle system.
For the lane change assistant, the *abstract function* f_A provides the *conformity oracle* with a single set of valid abstract actions. The *conformity oracle* checks if the abstract action from the *output abstraction* f_O is included in this set of valid actions. In case the abstract action from the *output abstraction* f_O is not included in the set, correct and safe behavior of the lane change assistant cannot be guaranteed.
For further improvements of autonomous vehicle systems, encountered system faults and corresponding information like critical situations and internal processing results have to be stored for later analysis. The logging of faulty system behavior and corresponding situation data is described in the Section 6.2.3.6. First, the implementation of *quantitative runtime monitoring* is presented in the next section.

6.2.3.5. Implementation Quantitative Monitoring

The *quantitative runtime monitoring* evaluates if encountered situations have already been known and verified in the development of the autonomous vehicle system. Encountered unknown situations are recorded for further improvements of autonomous vehicle systems. For this purpose, the *quantitative runtime monitoring* has two operating modes — *recording state* and *comparison state*:

Recording State: The *recording state* is primarily used in the verification of autonomous vehicle systems in order to record verified situations in system simulations as *tested situation knowledge*.

Comparison State: The *comparison state* is primarily used during operation of autonomous vehicle systems in the real world. The *situation oracle* compares encountered situations with the knowledge of verified situations. Unknown and unverified situations are identified and recorded because these unknown and unverified situations impose the potential danger for unsafe behavior by the autonomous vehicle systems.

The following sections describe the implementation of the situation recording for the *recording state* and the situation comparison for the *comparison state* in more detail.

6.2.3.5.1. Situation Recording

In the *situation recording*, the *quantitative runtime monitoring* accumulates the *tested situation knowledge*. Each processing cycle, the *situation monitor* receives the *abstract representations* of domain $\mathcal{D}_A^I$ from the *input abstraction* f_I. In the *recording state*, the abstract representation are recorded and added to the *tested situation knowledge* (cf. Fig. 5.1).
The two major decisions in the implementation of the situation recording for the *recording state* are storage and encoding:

Storage: Files and databases are two prevailing options for storage of the *tested situation knowledge*. Abstract representations from the *input abstract* f_I are either parsed into (text) files on the local hard drive or locally saved in a specialized situation database.

The segmentation of the storage space is another factor which has to be considered in the implementation of the situation recording. Files and databases can be instantiated to record *abstract situations* for multiple autonomous vehicle systems, for a single autonomous vehicle system, or a single session, e.g., a test suite or test case, with1 an autonomous vehicle system.

Available wireless network connections would allow storing files and database remotely, but a local cache would still be required in order to maintain the data integrity even for high latency and unreliable wireless connections.

Encoding: The encoding of the abstract representation in text files and databases may range from human readable, e.g., specific language, logic, or extensible markup language (XML), to unreadable formats, e.g., bytecode. Human readable formats support the manual analysis and validation of situation recordings but require larger disk space than unreadable formats.

Hashing could be used for a hybrid encoding approach which reduces the memory consumption. Instead of storing the complete data objects for each recorded situation, these situations are hashed, and their hash is stored. This way, the data of any recorded situation is only stored once and for any duplicates only the hash is stored repeatedly. The mapping of hashes to situations is stored in a central lookup file. This lookup file allows the manual analysis of the situation recordings.

For the lane change assistant, *abstract situations* from the *input abstraction* f_I are parsed into a XML file per test case using the BOOST serialization library (cf. [Sch11a]). The structure of the XML follows types and properties of the domain $\mathcal{D}_A^I$. The XML files are readable and analyzable by engineers. These XML files and their knowledge can be individually combined as *tested situation knowledge* for the situation comparison during operation in the real world.
Abstract situation representations may occur multiple processing cycles in a single simulation. Duplicate of situation representations do not improve the *tested situations knowledge* or the analysis of system behavior but increase the memory consumption of the *quantitative runtime monitoring*. For this reason, one single instance of each *abstract situation* is stored in the *tested situation knowledge*.
Every abstract representation from the input abstract f_I is compared to the existing situations in the *tested situation knowledge* and any duplicates are dismissed. Another check for duplicates will be performed in the situation comparison (see next section) if multiple files or databases are combined as *tested situation knowledge*.
The holistic engineering approach (cf. Section 4.3) requires all *abstract situations* of the *tested situation knowledge* to originate from simulations of successful and valid test cases. Otherwise, the correctness and safety of autonomous vehicle systems are evaluated by the *quantitative runtime monitoring* on critical and unsafe situations. For the *tested situation knowledge*, only *abstract situations* from successful and valid test cases are considered. In the case of failed test cases, autonomous vehicle systems and the test cases are improved until these test cases pass successfully.
The *recording state* of the *quantitative runtime monitoring* could be used during operation in the real world in order to gather realistic situations as input for the generation of the initial test scenarios and test cases for the autonomous vehicle systems. However, it is more appropriate to record these situations during operation in the real world in the *comparison state* with an empty *tested situation knowledge*. The *quantitative runtime monitoring* records all encountered situations because all situations are evaluated as unknown due to the empty set of *abstract situations* for the *tested situation knowledge*. Abstract situations recorded in the *comparison state* can directly be incorporated into the generation of new test scenarios and test cases (cf. Section 7.2.3). The format for the situation recording in the *recording state* and for the situation logging in the *comparison*

state must not match each other. However, the generation of test scenarios and test cases is tailored to the situation logging used in the *comparison state*. The implementation of the situation comparison of the *quantitative runtime monitoring* in the *comparison state* is described in the following section

6.2.3.5.2. Situation Comparison

In the *comparison state*, abstract representations from the *input abstraction* f_I are received by the situation monitoring in each processing cycle and forwarded to the *situation oracle* (cf. Fig. 5.1). The *situation oracle* compares the abstract representations with the *tested situation knowledge*. Abstract representations which are contained in the *tested situation knowledge* have already been considered in the verification of the autonomous vehicle system in simulations.
Unknown and unverified abstract representations which are not included in the *tested situation knowledge* are logged for the further improvement of the autonomous vehicle system.
In general, the *tested situation knowledge* for the *comparison state* has to be imported from potential multiple sources — files or databases (cf. Section 6.2.3.5.1). Known and verified situations may be stored in multiple text files or databases which have to be integrated for the *tested situation knowledge*. This segmentation allows to freely combine situation data from different simulation-based tests as *tested situation knowledge*. This way the *tested situation knowledge* can be optimized for the envisaged operation in the real world.
Wireless connections allow to import *tested situation knowledge* from external sources during operation. Bandwidth fluctuations of the wireless connections still require the local storage of the *tested situation knowledge* within the vehicle in order to avoid the time-consuming communication with external sources for the comparison of encountered *abstract situations*.
The import of the *tested situation knowledge* from multiple sources — text files and databases — may introduce duplicated *abstract situations*. These duplicates of *abstract situations* are eliminated in the import process of the *tested situation knowledge*. Duplicates offer no benefit for the *quantitative runtime monitoring* For the lane change assistant, the *tested situation knowledge* are imported from multiple XML files. In this import process, any duplicated *abstract situations* are removed.
The *situation oracle* compares each encountered abstract representation from the *input abstraction* f_I with the set of known and verified situations contained in the *tested situation knowledge*. The basic implementation evaluates at each processing cycle the equivalence of the encountered *abstract situation* and the *tested situation knowledge*. All properties values of each object from the encountered abstract representation are compared with corresponding objects in each *abstract situations* from the *tested situation knowledge*. For the lane change assistant, *abstract situations* are evaluated by the *situation oracle* for equivalence to the *abstract situations* in the *tested situation knowledge*. Other implementations may categorize encountered *abstract situations* based on probabilities (cf. [Agg14; MPB09]). Abstract situations which are not matched to any *abstract situations*

in the *tested situation knowledge* are judged as critical and potentially unsafe. These situations have to be analyzed and considered for further improvements of autonomous vehicle systems.
The *quantitative runtime monitoring* has a complexity $\mathcal{O}(n \cdot m)$ for the equivalence comparison of *abstract situations* where $0 < n < \infty$ denotes the number of abstract representations contained in the *tested situation knowledge* and $0 < m < \infty$ depicts the maximum number of objects per *abstract situation* which have to be compared for equivalence. The lower bound of n and m is 0 because negative numbers for situations in the *tested situation knowledge* and for objects in the *abstract situation* are impossible. Theoretically, there exist no upper limit for n and m. The numbers of objects in the vicinity of the autonomous vehicle system in the real world and numbers of situations in the *tested situation knowledge* are unlimited. In reality, the number of situations in the *tested situation knowledge* is limited by the available space on the data storage, e.g., the local hard drive. The number of objects in the vicinity of the automated ego vehicle is physically limited in the real world by the real dimensions in real dynamic objects and the maximum number of their instances within the sensor range of the autonomous vehicle system.
The *comparison state* of the *quantitative runtime monitoring* can be applied to the verification of autonomous vehicle systems in simulations. Based on the *abstract situations* from previous simulations as *tested situation knowledge*, the impact of new test cases can be evaluated. As a result, the previously unknown *abstract situations* which have yet been verified by the new test cases are highlighted.
The *quantitative runtime monitoring*, as well as the qualitative runtime monitoring, require the logging of *abstract situations* in case of hazardous situations reps. system behavior for further analysis and the improvement of autonomous vehicle systems. The implementation of the logging is described in the following section.

6.2.3.6. Implementation of Logging

For improvements of autonomous vehicle systems (cf. Section 4.3), the *qualitative* and *quantitative runtime monitoring* have to log information about the state of autonomous vehicle systems and their environments. The *qualitative runtime monitoring* logs *abstract situations* with incorrect and unsafe system behavior while the *quantitative runtime monitoring* logs *abstract situations* which have not yet been known and verified. Abstract representations of the domain $\mathcal{D}_A^I$ contain all necessary information about states of autonomous vehicle systems and their environments for the analysis of unsafe or faulty system behavior by engineers. The abstraction level of *abstract situations* matches the level of detail on which engineers intuitively reason about the behavior of autonomous system because the domain for the *abstract situations* are derived from requirements of the autonomous vehicle systems.
The single event in which incorrect and unsafe system behavior emerged is not sufficient for analysis of faults for autonomous vehicle systems. An *abstract situation* describes a single instance of the *system state* in which incorrect and unsafe system behavior emerged (cf. Definition 4.4). Engineers are interested in the sequence of events leading

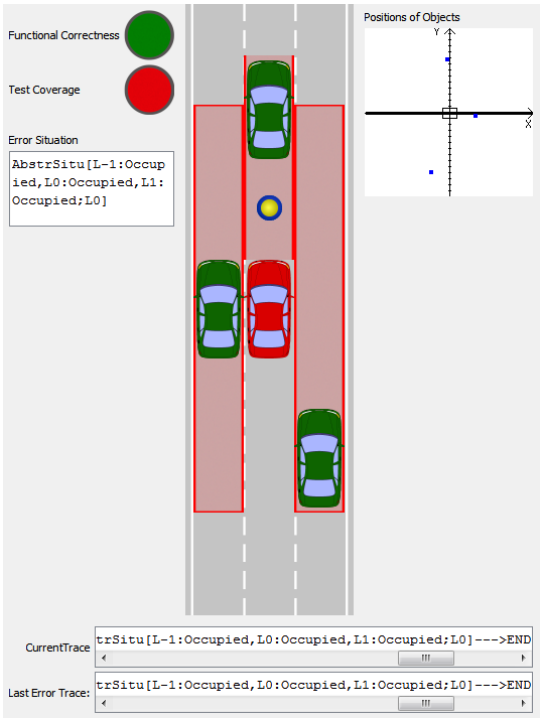


Figure 6.12.: Graphical visualization of an abstract situation.

up to this event. The sequence of events leading to the incorrect and unsafe behavior of autonomous vehicle systems has to be recorded as a viable *counter example* of the systems' correctness and safety. This history of *abstract situations* allows engineers to analyze the emergence of potential unsafe system behavior, localization of system faults, and the definition of new test cases (cf. Section 7.2.3).

The runtime monitoring framework implements a ring buffer in the *situation monitor* in order to record the history of *abstract situations*. The size of the ring buffer limits the maximum length of this history. Every processing cycle, the new abstract representation from the *input abstraction* f_I is appended to the ring buffer. In case the ring buffer is filled the oldest *abstract situations* is discarded.

The *qualitative* and *quantitative runtime monitoring* equally use the ring buffer of the *situation monitor* in the logging of critical *abstract situations*. In case the *quantitative runtime monitoring* encounters an unknown and unverified situation or the *qualitative runtime monitoring* detects incorrect and unsafe system behavior, the *abstract situation* and the content of the ring buffer are logged with a description about the detected fault as a *counterexample* in a log file. The formats of log files are subject to the implementation and may be human readable or not. Non-readable log files require additional tools for their analysis. For example, tools with a graphical visualization of *abstract situations* are beneficial for the analysis of either log files (cf. Fig. 6.12).

Log files are commonly stored locally on hard disks inside the vehicle. For analysis, engineers have to manually transfer the logs to the development computer at the garage by a portable hard disk or exchanging the hard disk in the vehicle. Vehicles equipped

with wireless network connections offer the possibility to transfer the log information directly to external storage.
The complexity of the data logging in the presence of a system failure or unknown situation has the complexity of $\mathcal{O}(n \cdot m)$. The complexity results from the length of the situation history n and the maximum number of objects m in all *abstract situations* of the history. In reality, the data encoding and the write speed of the hard disk drive (HDD) have a significant impact on the performance of the data logging. Complex encodings of the situations, e.g., human-readable formats, introduce additional data which has to be written to the HDD.
The *qualitative runtime monitoring* and *quantitative runtime monitoring* are deployed alongside autonomous vehicle systems in simulation-based verification and during operation in the real world. The application of the runtime monitoring framework in simulations of the system verification is described in the following section. The usage of the runtime monitoring framework during the operation of autonomous vehicles in the real world is presented in Chapter 7.

6.3. Monitor Training in Simulations

System simulations are used in the automotive domain for the verification of autonomous vehicle systems (cf. Section 3.6). As described in Section 5.2, autonomous vehicle systems are integrated into a simulation framework for verification. In the simulations, autonomous vehicle systems are evaluated for a defined set of test cases. System simulations offer the possibilities of faster development and verification cycles without having to costly setup prototype vehicles. In the holistic engineering approach for autonomous vehicle systems, the runtime monitoring framework is deployed alongside the autonomous vehicle systems in these simulation in order to evaluate the behavior of these systems (cf. Section 6.3.2) and to collect verified *abstract situations* for the *tested situation knowledge* (cf. Section 6.3.3).
The following sections address the utilization of the runtime monitoring framework in simulations. In Section 6.3.1, the definition, and execution of test cases in simulations for the verification of autonomous vehicle systems are described. Section 6.2.3.4.2, presents the utilization of the *qualitative runtime monitoring* as test oracle in the simulations. The accumulation of the *tested situation knowledge* in simulations is described in Section 6.3.3.

6.3.1. System Verification in Simulations

Autonomous vehicle systems are verified in system simulations based on models of their environment (cf. Section 3.6). Following the architecture from Section 5.2, information from these environment models is processed by sensor models and the system's *preprocessing* as test inputs for the verified *function*. Environment models and have to sufficiently represent traffic situations which autonomous vehicle systems will encounter during operation in the real world. This representation includes the scenery of the static elements, e.g., roads, signs, and buildings, as well as dynamic objects, e.g., vehicles and

pedestrians. Any sparse representations of real-world situations diminish the validity of corresponding verification results for the operation of autonomous vehicle systems in the real world. Following the *open-world assumption*, the real world is impossible to be comprehensively modeled and by a single environment.

Definition 6.2 (Environment Model)**.** *An environment model represents the real world in a simulation. The model includes the scenery of static objects, e.g., roads, signs, as well as dynamic objects, like vehicles and pedestrians and their behavior.*

Abstract modeling of the environment is necessary for system simulations in order to sufficiently generate realistic test inputs for the autonomous vehicle system. Modeling of the vehicle's environment as an environment model is vital in order to generate test inputs for the autonomous vehicle system sufficiently. The behavior of autonomous vehicle systems may slightly deviate in repeated executions of tests. Each test would require slightly different test inputs. The usage of recorded system signals directly as test inputs for the autonomous vehicle systems without any abstraction would not be able to provide these slightly deviating test inputs.
In system verification (cf. Section 3.6), system parts or complete autonomous vehicle systems are evaluated for a set of simulation-based tests (cf. Section 2.2). Simulation-based tests are defined in a two-stage process as *test scenarios* and *test cases*. Test scenarios define static elements of the scenery and the behavior of dynamic objects in the simulated world for a limited duration. For the scenery, the course of the road including its lanes and markings are defined. Additional beautifications like vegetation or building can be added to the scenery. Dynamic vehicles, e.g., vehicles and pedestrians, are positioned in the world based on the course of the road and their behavior is defined. For the ego vehicle, test scenarios define a target destination which the ego vehicle has to reach at the end of the scenario. Positions of static and dynamic objects as well as behaviors of dynamic objects are modeled to cause the anticipated behavior, e.g., lane changes, by the autonomous vehicle systems on its way towards its target destination.
Test scenarios do not define all parameters which are required by the simulation. Some parameters are not yet explicitly defined in test scenarios and offer the possible for parameter fuzzing (cf. [Ana+13; Bac+17c; MP+07]). For example, the scenario for a lane change can be tested for different velocities of the ego vehicle.
Test cases extend test scenarios by defining explicit values for the free parameters of test scenarios, e.g., velocities and acceleration of vehicles. With the specification of free parameters, the expected behavior of the autonomous vehicle systems for the test cases is defined. Multiple test cases can be defined for one single test scenarios. Each test case may expect different behavior from the autonomous vehicle system under test. All test cases for the verification of an autonomous vehicle system are aggregated to a test suite.
Test scenarios and test case are implemented as environment models. These environment models are either manually modeled by engineers from the system's requirements or derived from recorded traffic situations during operation (cf. Section 7.2.3). Static and dynamic objects are positioned in a virtual world, and their behavior and parameters are configured according to the test cases.

The case study (cf. Chapter 8) has shown that engineers tend to model the most straightforward scenarios for the verification of particular system behavior, e.g., a lane change, but these scenarios occur in limited quantities in the real world. For example, one single vehicle is modeled in front of the autonomous vehicle in order to trigger a lane change by automated ego vehicle. Scenarios in the real world tend to be far more complex. Larger quantities of vehicles drive alongside the autonomous vehicle on public roads than anticipated by engineers. The holistic engineering approach addresses this issues by transferring information from the real world back to the development for the definition of realistic test scenarios and test cases (cf Section 7.2.3).
The simulation framework successively executes the test suite for an autonomous vehicle system. Each test case commences with the configuration of internal system parameters and initialization of the environment model. In the execution of these simulation-based tests, dynamic objects of the environment models perform their assigned behaviors. The autonomous vehicle system operates in this virtual environment based on the inputs about the virtual world from *sensor models* and system *preprocessing*. The system processes maneuvers for its immediate and future driving direction towards its assigned destination (cf. Section 3.3.1.3). The output of the system *function* — the driving maneuver — is considered by the simulation framework to change the position and pose of the autonomous vehicle in the simulated world. Depending on the type of the simulation, other dynamic objects are influenced by the behavior of the autonomous vehicle (closed-loop simulation) or operate independently — solely based on the defined behavior by the test case (open-loop simulation).
For the lane change assistant, eight test scenarios have been modeled based on the requirements for the lane change assistant (cf. Section 3.2). Each test scenario has been parameterized by at least two test cases (cf. Appendix A.2). The environment model for each test cases has been modeled in VTD by manually defining the road structure, road topology and placing necessary dynamic vehicles on the road (cf. [Oli+16; Ulb+16]). The behavior of these vehicles has been defined considering the expected behavior of the lane change assistant for the test case. Overall 24 test cases have been manually designed (cf. Chapter 8). For further information about configuration, execution, and results of the simulations for the verification of the lane change assistant can be found in (cf. Chapter 8).
The correctness and safety of emerging system behavior in the simulations are evaluated manually by engineers or automatically by a test oracle in regard to the expected behavior for the autonomous vehicle systems by the test cases. Maneuvers performed by the automated ego vehicle have to correspond to the maneuvers which have been anticipated by the test cases. Test oracles evaluate the behavior of autonomous vehicle systems based on a set of given criteria and conditions. As shown in the following Section 6.3.2, the runtime monitoring framework can be used in simulation as test oracle.

6.3.2. Qualitative Monitoring as Test Oracle

The *qualitative runtime monitoring* can be used as *test oracle* (cf. [Bar+15]) in simulations in order to support the verification of autonomous vehicle systems. As *test oracle*, the

qualitative runtime monitoring evaluates the correctness and safety of the system's behavior concerning conditions which are defined by the system requirements. However, the correct execution of maneuvers by the autonomous vehicle system in comparison to the anticipated behavior by the test cases is not assessed by the qualitative monitoring. For example, a test case for a lane change may be evaluated as correct and safe by the *qualitative runtime monitoring* even though the automated ego vehicle remains within its current driving lane without performing the anticipated lane change. Engineers have to manually evaluate the actual system behavior under consideration of the intended behavior for the autonomous vehicle system in the test case.
A test case for the verification of autonomous vehicle systems will be judged correct and safe if the *conformity oracle* evaluates the abstracted action of the autonomous vehicle system as correct and safe for each (abstract) situation emerging in the simulation of the test case. A simulation can be seen as a trace of (abstract) situations (cf. Definition 2.32). Each processing cycle, the *conformity oracle* compares the abstracted action of the autonomous vehicle system with the sets of abstract actions from the *abstract function* f_A (cf. Section 6.2.3.4). Equation (6.5) must yield.
As a result, a test case for the autonomous vehicle system will be successful if the *qualitative runtime monitoring* and the test engineer evaluate the behavior of the autonomous vehicle system in the simulation as correct and safe. The manual and automatic evaluation of the test case represents the *ground truth* for the usage of any results from the simulation (cf. Section 6.3.3) as references for the evaluation of the autonomous vehicle system during operation in the real world. Any malicious test case requires the manual validation and analysis of engineers. Engineers have to determine if the faulty test case has been raised by faults of the autonomous vehicle systems, faulty test cases, or faults within the qualitative runtime monitoring.
For the soundness of the qualitative runtime monitoring, the *qualitative runtime monitoring* and its verdicts have to be validated. *Input abstraction* f_I, *output abstraction* f_O, *abstract function* f_A, and *conformity oracle* are implemented based on the typed first-order logic (cf. Section 6.2.3.4) and may include implementation faults. For example, Eq. (6.9) could be violated for the *input abstraction* f_I and *output abstraction* f_O. Under the assumption that the autonomous vehicle system has been correctly implemented and the manual verdicts by engineers are sound, simulations offer the possibility to validate the *qualitative runtime monitoring* by comparing its verdicts to the manual evaluation of the system's behavior by engineers in each situation. System requirements have to be faulty and have to be revised if the deviating verdicts do not originate from implementation faults.
For the lane change assistant, success and failure of test cases in the system verification have been determined manually by test engineers. The *qualitative runtime monitoring* has been implemented based on the requirements for the lane change assistant (cf. Section 3.2) and has been used in the simulations of the system verification as an additional evaluator. This approach allowed to validate the verdicts of the *qualitative runtime monitoring* for test cases based on the manual verdicts by engineers for these tests. The *qualitative runtime monitoring* evaluated two test cases for the lane change assistant as correct and safe even though the automated ego vehicle did not excerpt the anticipated behavior for

these test cases. These two test cases were manually evaluated as unsuccessful. More details about the application of the *qualitative runtime monitoring* in simulations for the lane change assistant are given in Chapter 8.
While the *qualitative runtime monitoring* as a *test oracle* evaluates the correctness and safety of the autonomous vehicle system in the simulations, the *quantitative runtime monitoring* uses the simulations for the recording of its *tested situation knowledge* for later use during the operation of the autonomous vehicle system in the real world. The usage of *quantitative runtime monitoring* in the verification of autonomous vehicle systems is described in the following section.

6.3.3. Training of Situation Monitor

In simulations, the *quantitative runtime monitoring* is used to gather the *tested situation knowledge*. The execution of test cases in simulations results in a trace of situations where each situation describes a concrete configuration of the virtual world at a given point in time (cf. Definition 2.32). The *quantitative runtime monitoring* records each *abstract situation* from the *input abstraction* f_I. Encountered *abstract situations* are stored either in a file or a database (cf. Section 6.2.3.5). Any duplicates of encountered *abstract situations* are dismissed leaving a set of *unique abstract situations* (cf. Definition 6.3). For the recording of situations, the *quantitative runtime monitoring* has to operate in the *recording state* (cf. Section 6.2.3.5).

Definition 6.3 (Unique Abstract Situation)**.** *A unique abstract situation is a* abstract situation *in a set of abstract situations which has no duplicates within this set.*

All recorded *abstract situation* compose the *tested situation knowledge* which only contains verified *abstract situations*. It is assumed that all test cases for an autonomous vehicle system are successful and can be used in the recording of the *tested situation knowledge*. The manual evaluation by engineers and the automatic evaluation by the *qualitative runtime monitoring* of test cases (cf. Section 6.3.2) represents the ground truth for the correctness and safety of the recorded *abstract situation*. Any unsuccessful test case would require the analysis and improvement of the test case or autonomous vehicle system until the test case has been successfully executed. The *abstract situations* of the *tested situation knowledge* are used as the reference set for the *quantitative runtime monitoring* of the autonomous vehicle system during operation in the real world (cf. Chapter 7).
The recording of *abstract situations* by the *quantitative runtime monitoring* enables statements about the impact of individual test case for *tested situation knowledge*. The number of *unique abstract situations* for a test case can be used as an indicator of its complexity. The comparison of the set of recorded *abstract situations* for a test case with the set of *abstract situation* in *tested situation knowledge* describes the impact of individual test cases for the *quantitative runtime monitoring* The knowledge about the impact of individual test cases in term of *unique abstract situations* can support the definition of a reduced but impactful subset of test cases for an autonomous vehicle system in order to optimize its verification. For example, a subset could be defined

which engineers should initially use for the verification and validation of their system improvements.
The runtime monitoring has recorded 734 *abstract situations* have been recorded for the 24 initial test cases in XML files in the case study on the lane change assistant. The XML files of the 22 valid test cases are imported as *tested situation knowledge* for the *quantitative runtime monitoring* of the lane change assistant during operation (cf.Section 7.1.2). The recordings from the two unsuccessful test cases are omitted in the *tested situation knowledge*. The statistics about recorded *abstract situations* for individual test cases have also been evaluated and are presented in Chapter 8.
The *quantitative runtime monitoring* during operation in the real world is described in Chapter 7.
First, the impact and limitations of the runtime monitoring framework are discussed in the next section.

6.4. Impact and Limitations of the Runtime Monitoring Framework

The exclusive runtime verification of system properties is not sufficient for the reasoning about the correctness and safety of autonomous vehicle systems in the real world. As described in Section 3.8.1, the real world is too complex to fully specify and to verify all possible input situations for the *function* of the autonomous vehicle system. The *qualitative runtime monitoring* only verifies the relationship between the input situations and the output action of the *function* but not the validity of input situations themselves. Therefore, input situations may occur which have not yet been considered in the implementation of the function and are outside its definition range. The processing of the output action by the *function* might still be correct and valid but not be valid and safe for the situation in the real world. Therefore, it is necessary to monitor and verify the environment situations. The *quantitative runtime monitoring* addresses this issue by recording the unknown situations in order to verify the behavior of autonomous vehicle systems for the situations in simulations. Revising the runtime monitoring in the real world and simulation as proposed by the engineering approach of this work, the set of verified and safe situations for the autonomous vehicle system continuously increases.
A fully verified and functional runtime monitoring framework should not have any *false negative* in regard to the system requirements (cf. Table 3.12). *False negatives* for the runtime monitoring are critical situations or critical system behavior which are not recognized by the runtime monitoring framework. However, this restriction does only apply to considered requirements. Manual evaluations of simulations and test drives in the real world may reveal *false negative* which are subject to the underspecification of the requirements and which cannot be detected by the runtime monitoring. The completeness of the requirements is essential for the quality of the runtime monitoring.
The runtime monitoring framework does record *false positives* (cf. Table 3.12). The *quantitative runtime monitoring* may encounter situations during operation in the real

world which it classifies as unknown and unverified but in which the behavior of the autonomous vehicle system in simulation is correct and safe. Under the assumption that the *input abstraction* f_I, *output abstraction* f_O, *abstract function* f_A, and *conformity oracle* are implemented correctly, *false positives* for the *qualitative runtime monitoring* indicate inadequacies in the system requirements.
The validity of the runtime monitoring framework is restricted by the considered requirements and the soundness of *input abstraction* f_I and *output abstraction* f_O. The scope of considered requirements directly defines the scope of the runtime monitoring framework. Even though results from the runtime monitoring framework can be used to enhance the specification (cf. Section 4.3.3), the runtime monitoring is only able to consider objects and properties of systems and their environments in its *abstract situations* which are defined in the requirements for the autonomous vehicle systems. Other objects and properties are omitted from the *abstract situations* and are not considered by the *qualitative* and *quantitative runtime monitoring*.
As described in Section 6.1.4, the *input abstraction* f_I and *output abstraction* must not process situations identically for which the autonomous vehicle systems exhibit different behaviors. Otherwise, any result from the runtime monitoring framework is assumed to be incorrect. For this reason, the implementation of the runtime monitoring framework and its results have to be validated within the system verification by test engineers based on their manual evaluation of test cases. The runtime monitoring framework should only be used unsupervised as test oracle in simulations if the soundness of the runtime monitoring framework has been sufficiently validated.
Data abstractions have a long history in model checking and verification [CGL94]. Nevertheless, abstractions are considered by other approaches (cf. [Kan15; KK15]) but are not explicitly defined and emphasized in the same way as in this work. These approaches (cf. [Kan15; KK15]) monitor the correctness of systems by conditions which are defined directly for the data objects and signals of monitored systems. Such a limited level of abstraction only allows highly technical engineers the analysis of monitoring results. The abstraction of the runtime monitoring framework reduces the overall data complexity by its discretization of the real world. The real world and its complexity require the abstraction not to be too coarse and not to be too detailed (cf. [Den+14]). The consideration of requirements for the definition of *input abstraction* f_I and *output abstraction* f_O result in an overall abstraction by the runtime monitoring framework on the level of the system specification. Correctness and safety of autonomous vehicle systems have not to be monitored directly on system objects and system signals but are evaluated on abstract data which matches the level of abstraction on which engineers reason about the system behavior. This abstraction supports the manual analysis of runtime monitoring results and the identification of faults for the autonomous vehicle systems. The complexity of the runtime monitoring can be further improved by considering domain knowledge, e.g., the limited range of sensors (cf. [Bar+09]), for the domains of objects and values in the abstract representation.
The *quantitative runtime monitoring* enables the reasoning about the correctness and safety of autonomous vehicle systems based on the number and type of encountered and verified situations. This information can be integrated in metrics for the correctness and

safety of autonomous vehicle system (cf. Section 3.8.4). Such metrics would be more expressive and sound than current metrics for autonomous vehicle systems. Current metrics solely evaluate their safety based on the number of faults over the total driving distance of autonomous vehicle systems. Existing metrics judge autonomous vehicle systems as correct and safe for all possible environment situations even if systems have just been driven and verified on a straight rural road with limited traffic. Metrics incorporating numbers and types of situations would explicitly consider and address the diversity of real-world situations.

While Chapter 6 presented the implementation of the runtime monitoring framework and its usage in the system verification, Chapter 7 addresses the operation analysis by applying the runtime monitoring framework to the operation of the autonomous vehicle systems in the real world. The results from the runtime monitoring during operation enable the system evolution and definition of test scenarios and test cases for dependability improvements of autonomous vehicle systems (cf Section 4.3.3).

7. Operation Analysis and System Evolution

While Chapter 5 addressed the architecture of the runtime monitoring framework and Chapter 6 presented the conception, implementation, and training of the runtime monitoring framework, this chapter describes the application of the runtime monitoring framework during operation of autonomous vehicle systems in the real world as well as the usage of its results for improvements of autonomous vehicle systems. As shown in Fig. 4.1, the application of the runtime monitoring framework during operation is the third part of the engineering approach. The runtime monitoring during operation concerns questions about the validity and soundness of verification results during operation in the real world and the improvement of autonomous vehicle systems by the result of the runtime monitoring during operation.
As in the simulations of the system verification (cf. Section 6.3), the runtime monitoring framework is deployed alongside the autonomous vehicle system inside the vehicle. Results from the runtime monitoring during operation in the real world can be used in order to support the safety of the autonomous vehicle systems during operation and for the improvement of these systems in additional development cycles. The runtime monitoring and safety enforcement during operation are described in Section 7.1. The improvement of autonomous vehicle systems based on the runtime monitoring results is described in Section 7.2. Section 7.2.3 addresses the generation of new test cases from recorded *abstract situations*.

7.1. Runtime Monitoring at Operation

The runtime monitoring framework and its components (cf. Fig. 5.1) are identically used for runtime monitoring of autonomous vehicle systems during operation in the real world as they are used in simulations of the system verification (cf. Section 6.3). The *qualitative runtime monitoring* assesses the system behavior of autonomous vehicle systems based on the systems' requirements while the *quantitative runtime monitoring* evaluates encountered real-world situations. The main differences in the usage of the runtime monitoring framework in simulations of the system verification and during operation in the real world are

- the change from sensor and actuator models to real sensors and actuators, and
- the usage of verified *abstract situations* from simulations of the system verification as *tested situation knowledge*.

In comparison to the modeled world in simulations, the real world is far more complex with an uncountable number of possible traffic situations. All possible traffic situations can never be fully simulated (cf. Section 2.2). All dynamic objects in the real world act as autonomous agents (cf. [FG96]). Though dynamic objects might react to the behavior of the automated ego vehicle, their actions and reactions are not predefined nor are they deterministic. Neither the autonomous vehicle system nor system engineers have control over the behavior of other objects in the real world. An autonomous vehicle system requires physical sensors for the perception of its environment. Sensors are crucial for autonomous vehicle systems, because communication service networks for the communication with vehicles and infrastructure, e.g., V2X, are not extensively available and are not support by all environmental objects, e.g., houses, tree, or animals.
For the perception of objects in the real world, the autonomous vehicle system and the runtime monitoring framework have to be installed in real vehicles alongside real sensors and actuators. For the runtime monitoring of autonomous vehicle systems during operation in the real world, two use cases can be distinguished :

Field tests In the development of the autonomous vehicle systems, prototype vehicles are used in field operational tests to verify and validate the behavior of these systems. The focus of field testing is on the validation of the system implementation and its final configuration for production vehicles. Field testing is the last activity in the development process of autonomous vehicle systems in which car manufacturers can identify and resolve system failures before the autonomous vehicle systems are installed in production vehicles and are used by customers (cf. Fig. 3.2).

Field tests may take place on test tracks or public roads. On test tracks, safety and performance are tested by prototype vehicles in critical traffic situations without any danger for other traffic participants[1]. Field tests on public roads are used for the testing of the system's behavior in realistic environments to gain an outlook on the system behavior during operation by customers.

Operation by customers production vehicles as the final product are bought and operated by customers in their day-to-day lives. Customers are less sophisticated with the capabilities and limitations of autonomous vehicle systems than test engineers. Production vehicles and their autonomous vehicle systems have to be sufficiently verified and validated in the vehicle's development before the usage by costumers. Customers use production vehicles more frequently and extensively than test engineers in field operational tests. Therefore, customers achieve higher mileage and encounter a more extensive variety of traffic situations.

The larger mileage by customers would enable car manufacturers to gather larger quantities of realistic data about the systems' behavior and real traffic situations by the runtime monitoring. The high amount of traffic situations would be beneficial for the development of existing and future autonomous vehicle systems. However,

[1] https://eu.usatoday.com/story/tech/2017/10/31/waymo-self-driving-cars-go-school-here/815627001/ (accessed: 12/02/2018)

> data from production vehicles is insufficiently considered by the car manufactures for the development of autonomous vehicle systems. Nowadays, only Tesla Motors is known for recording massive amounts of real-world data from its autopilots in production vehicles [2].

Autonomous vehicle systems in prototype vehicles and production vehicles primarily deviate from systems in simulations of the system verification in the usage of physical sensor and actuators instead of sensor and actuator models. As shown in Fig. 5.1, the change to physical sensor and actuators results in changes for the *preprocessing* resp. *postprocessing*.
Physical sensors are prone to provide sensor data with jitter that may lead to the potential faulty mapping of real-world objects in the internal representation of the real world (cf. Section 3.8.1). Physical sensors, actuators, and their corresponding processing have to be additionally verified by complementary V&V methods. The runtime monitoring framework does not address the verification and validation of system components other than components which are contained within the *function* (cf. Fig. 5.1).
The use cases — prototype vehicle and production vehicle — have an impact on the quality and quantity of generated runtime monitoring data as well as the availability of recorded data for analysis and system improvement. As shown in Fig. 7.1, fields tests allow engineers to access the recorded data within minutes, hours, or days. However, the quality and quantity of the data are limited due to the shorter distances driven in field operational tests.
The operation by customers leads to larger amounts of realistic data, but this data can only be accessed in a workshop during vehicle maintenance. The regular intervals for maintenance range from month to years (cf. Fig. 7.1).
Wireless connections can enhance the access to runtime monitoring data in production vehicles. These wireless connections allow car manufacturers to access the data in production vehicles directly and to transfer them to their data storages. Until recently, neither car manufacturers nor customers have been willing to pay for the additional costs for wireless connections in production vehicles. An exception is Tesla Motors. Tesla Motors exhaustively use wireless connections to gather data from production vehicles in public traffic as well as update the vehicles' software[3]. The following sections describe the runtime monitoring during operation in the real world in general — applicable to both field operational tests and the operation by customers.
The runtime monitoring framework (cf. Fig. 5.1) can be used identically in prototype vehicles and production vehicles during operation in the real world as it has been used in the simulation of the system verification (cf. Section 6.3) however, the *function* must not change between system verification and operation in the real world. Valid argumentations about the correctness and the safety of the *function* require runtime monitoring results from simulations of the system verification and monitoring results from the operation in the real world to target the same *function*. The ground truth for the correctness

[2]https://electrek.co/2017/05/06/tesla-data-sharing-policy-collecting-video-self-driving/ (accessed: 12/01/2018)
[3]http://fortune.com/2015/10/16/how-tesla-autopilot-learns/ (accessed: 12/02/2018)

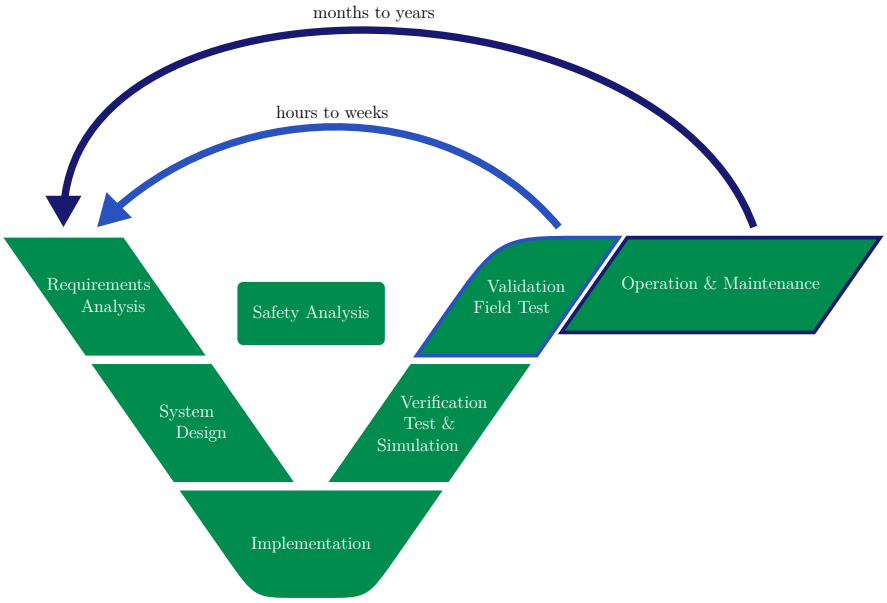


Figure 7.1.: Time duration until recordings are available for system evolution.

and safety of autonomous vehicle systems from the simulations will not be valid for operation in the real world if two different *functions* are monitored in simulations and during operation. The behavior of the *function* during operation in the real world has not yet been verified in simulations for any traffic situations.

The defined data interfaces have to remain unchanged and accessible by the runtime monitoring framework (cf. Section 5.3.1). However, the technical realization of these interfaces inside prototype vehicles and production vehicles may change from, e.g., communication via shared memory in simulations to message passing via vehicles' CAN buses.

Besides physical sensors and actuators, the operation mode of the *quantitative runtime monitoring* and its usage of the *tested situation knowledge* is the second main difference between the runtime monitoring framework in simulations of the system verification and during operation in the real world. During operation, the *quantitative runtime monitoring* operates in the *comparison state*. Verified and known *abstract situation* from the simulation are used as *tested situation knowledge* for the evaluation of encountered *abstract situation* (cf. Section 6.2.3.5). Unlike in simulations of the system verification, no new traffic situations are recorded for the *tested situation knowledge*. The *quantitative runtime monitoring* during operation in the real world is described in more detail in Section 7.1.2.

The autonomous vehicle systems during operation in the real world will be judged as safe for the current traffic situation by the runtime monitoring framework if the *qualitative runtime monitoring* evaluates the system behavior in the current traffic situation as safe and the *quantitative runtime monitoring* evaluate the current situations as known

and verified. The *qualitative runtime monitoring* must not recognize any unsafe action by the *function*. The *quantitative runtime monitoring* must find the current traffic situation within its *tested situation knowledge*. In any other case, the runtime monitoring framework cannot guarantee the correctness and safety of autonomous vehicle systems. Nevertheless, the autonomous vehicle system may behave safely in traffic situations that have not yet been known and verified.
The runtime monitoring framework only identifies traffic situations which have not yet been verified and in which the autonomous vehicle system behaves unsafe. These verdicts of the *qualitative* and *quantitative runtime monitoring* can be used during operation in the real world for the initiation of safety measures in order to maintain the vehicle's safety in critical situations. However, the selection and execution of safety measures in the presence of unsafe traffic situations are not addressed by the runtime monitoring framework. The selection and execution of safety measures is a complex problem (cf. [Hör11; RM15]). Multiple safety measures can be applied in a single critical situation. Possible safety measures may include but are not limited to

- transferring the vehicle control back to the driver in reasonable time,
- activating fall-back components in order to replace malicious parts of the autonomous vehicle system, or
- activating a fall-back routine with reduced safety functionality in order to take over the vehicle control and transfer the automated ego vehicle into a safe state. Such fall-back routines may include e.g.
  - an immediate full stop, or
  - the change to the emergency lane for a safety stop.

One of the most intuitive safety measures would be the initiation of a full stop. However, a full stop on highways is potentially hazardous. In situations where the automated ego vehicle is followed by another vehicle in short distance at high speed on a highway, a full-stop could potentially lead to a collision with the following vehicle. The following vehicle might be unable to react timely in order to avoid the collision. Therefore, the selection and execution of safety measure is not part of this work. The reader is referred to other work in this field, e.g., [Hör11; RM15].
The following sections give a more detailed description about the *qualitative* and *quantitative runtime monitoring* during operation in the real world. The *qualitative runtime monitoring* is described in Section 7.1.1. Section 7.1.2 describes the *quantitative runtime monitoring* during operation in the real world.

7.1.1. Qualitative Evaluation and Safety Enforcement

The *qualitative runtime monitoring* during operation supervises the behavior of the autonomous vehicle systems by evaluating the processing of the system's *function* regarding its requirements.

Table 7.1.: Implications for results from the qualitative runtime monitoring.

Dependable	Erroneous	Timing	Conclusion
Function Beh. ≡ Defined Beh.	Function Beh. ≠ Defined Beh.	Online	⇒ Initiate Safety Measure
		Offline	⇒ Revise Implementation

The conformity oracle will evaluate — as already shown for the simulation of the system verification (cf. Section 6.3.2)) — the behavior of the *function* for the current traffic situation as correct and safe if the abstract action from the *output abstraction* f_O matches the set of *abstract actions* by the *abstract function* f_A (cf. Table 7.1). Equation (6.5) has to yield. The *abstract function* f_A processes a set of correct and safe *abstract actions* resp. a set of illegal actions based on the abstract representation from the *input abstraction* f_I (cf. Section 6.2.3.4.1) in each processing cycle.

The behavior of the complete autonomous vehicle system can be judged as correct and safe during operation in the real world if all other parts of these systems which are not supervised by the runtime monitoring framework operate correctly and safely. As described in Section 5.2, these remaining parts have to be verified, validated, and monitored by other V&V methods.

The correctness and safety of the autonomous vehicle system will not be guaranteed by the runtime monitoring framework, if the abstract action from the *output abstraction* f_O does not match the abstract actions by the *abstract function* f_A. Misbehavior of the autonomous vehicle system imposes a danger for the safety of the automated ego vehicle, its passengers, and all objects and people in the vehicle's vicinity. As shown in Table 7.1, safety measures have to be initiated in the presence of incorrect system behavior in order to mitigate the safety risks by the malicious autonomous vehicle system *online* during operation in the real world. As mentioned in the previous section the selection and execution of the correct safety measure are beyond the scope of this thesis.

The resolution of faulty system behavior requires the improvement of the autonomous vehicle system offline by further development activities (cf. Section 3.1.2). For the analysis, identification, and improvement of incorrect system behavior, the *abstract situations* in which the malicious system behavior emerged are recorded in log files (cf. Fig. 5.1). The log files about incorrect system behavior include the type of malicious system behavior, the current *abstract situation* and the history of previous *abstract situation*. Depending on the use case — prototype vehicle or production vehicles —, the availability may range from days and weeks to month and years (cf. Fig. 7.1). A more detailed description of the system evolution is given in section Section 7.2.

The *abstract function* f_A and the *conformity oracle* f_C use the identical implementation in the simulations of the system verification and during the real world operation (cf. Section 6.2.3.4). This implementation does not include the selection and execution of safety measures in present of faulty system behavior. In the presence of faulty system

behavior, information about faulty system behavior, corresponding *abstract situations*, and the history of previous *abstract situation* are recorded in human-readable text files. The log files contain sequences of *abstract situations* with quite identical histories of previous *abstract situation*. These log files are essential for the improvement and evolution of the lane change assistant (cf. Section 7.2).
The *qualitative runtime monitoring* recorded estimated 29 unique *abstract situation* with incorrect and unsafe system behavior in field operational tests with a prototype vehicle on the German highway A2 (2.7% of 1078 unique situations) and estimated 19 unique *abstract situation* for the test drive on the German highway A39 (1,98% of 974 unique situations). The detailed statistic is clarified in Chapter 8.
Alongside the *qualitative runtime monitoring* of autonomous vehicle systems, encountered situations are evaluated by the *quantitative runtime monitoring* during operation in the real world. The *quantitative runtime monitoring* during operation in the real world is described in the following section.

7.1.2. Quantitative Evaluation and Situation Recording

The *quantitative runtime monitoring* during operation in the real world does not record encountered *abstract situation* for the *tested situation knowledge* but compares encountered *abstract situation* with the *abstract situation* in *tested situation knowledge* (cf. Section 6.2.3.5). The components used for the *quantitative runtime monitoring* during operation in the real world are identical to the components used for the *qualitative runtime monitoring* in simulations of the system verification. However, the functionality of the *quantitative runtime monitoring* deviates. During operation in the real world, the *quantitative runtime monitoring* operates in the *comparison state*.
At startup, *abstract situations* which have been previously recorded in simulations of the system verification are imported from files or databases for the *tested situation knowledge* during operation. The *tested situation knowledge* only contains *unique abstract situations* because all duplicates of *abstract situations* are eliminated in the import process. All *abstract situations* of the *tested situation knowledge* must originate from successful simulations with correct and safety system behavior by the autonomous vehicle systems. The correct verification of the autonomous vehicle systems in the simulation represents the ground truth for the runtime monitoring for these systems during operation in the real world.
During operation in the real world, each encountered traffic situation is transformed into an *abstract situation*by the *input abstraction* f_I. The *situation monitor* receives the *abstract situation* and adds it to its history of *abstract situations*. The *abstract function* f_A forwards the *abstract situations* to the *situation oracle*. The *situation oracle* compares the *abstract situation* with the situations contained in the *tested situation knowledge*.
In case the encountered *abstract situation*is not contained in the *tested situation knowledge*, the autonomous vehicle system has not yet been verified for this situation, and the correctness and safety of the autonomous vehicle system cannot be guaranteed — even if the *qualitative runtime monitoring* evaluates the behavior of the autonomous vehicle system as correct and safe.

Table 7.2.: Implications for results from the quantitative runtime monitoring.

Dependable	Erroneous	Timing	Conclusion
Situation ∈ Sit. Knowledge	Situation ∉ Sit. Knowledge	Online	⇒ Initiate Safety Measure
		Offline	⇒ Revise Test Cases

The safety argumentation for the operation of autonomous vehicle systems in the situation is incomplete because the ground truth from the simulations of the system verification is missing in the situation. The autonomous vehicle system might operate differently in unknown traffic situations than engineers would expect these systems to do. The actual behavior of autonomous vehicle systems need not be safety-critical. For example, the lane change assistant could follow a preceding slower vehicle on a multi-lane road instead performing a lane change for advantageous overtaking of the preceding vehicle.

While the *qualitative runtime monitoring* reveals safety critical system behavior with immediate danger for the vehicle, its passengers, and all objects and passenger in the vehicle's vicinity, the autonomous vehicle system may still operate correctly and safely in the presence of unknown and unverified traffic situations. The autonomous vehicle system need not impose a safety threat for the vehicle, its passengers and other objects and person in the vehicle's vicinity.

As for the *quantitative runtime monitoring* safety measures should be initiated during operation in the real world in order to avoid any potential danger if unknown traffic situations emerge. However, the criticality in an unknown and unverified traffic situation might not be as high as for malicious system behavior. It might be sufficient to alert the driver to monitor and evaluated the current traffic situations and to intervene if necessary. Other safety measures may still be applied.

Abstract situations which have not yet been known and verified are recorded for the extension of the ground truth in additional simulations and the improvement of autonomous vehicle systems. Each unknown *abstract situation*is stored in a log file or database with the history of previous *abstract situation* and additional meta information, e.g., time (cf. Section 6.2.3.5). In case a wireless connection is available, the data can immediately be transferred to remote storages.

Even though histories of previous *abstract situation* for successive unknown *abstract situation* contain large numbers of identical *abstract situation*; these histories are all stored individually in the log file. The primary usage of these log files in the engineering approach is the definition of additional test scenarios and test cases for the simulations of the system verification in further system development (cf. Table 7.2). The additional tests help to reveal additional faults for autonomous vehicle systems and to extend the scope of traffic situations with ground truth.

The recording of *abstract situations* by the *quantitative runtime monitoring* during operation in the real world enables statements about the impact of tests on a test track

or in public traffic. The number of unknown *abstract situation* which are not contained in the *tested situation knowledge* is an indicator of the impact of each test drive. Tests with higher amounts of unknown *abstract situation* are more likely to reveal previously unknown faulty system behavior and therefore are more likely to enhance the safety of the autonomous vehicle system. The information about the impact of test drives can be used to support test engineers to perform the validation of the autonomous vehicle system in the real world more effectively.

For the lane change assistant, the 77 recorded situations from successful simulations of 22 test cases in system verification are aggregated for the *tested situation knowledge* during operation in the real world. In test drives (cf. Chapter 8), each encountered *abstract situation* is compared to the *abstract situation* of the *tested situation knowledge* and unknown and unverified *abstract situation* are recorded with their histories of previous *abstract situations* in human-readable text files individually for each test drive.

As shown in Chapter 8, less than 1.6% of encountered *abstract situations* in the recordings of test drives on the German highways A2 and A39 have been tested in the simulations of the manually modeled test cases. This low percentage of coverage indicates insufficient modeling of the real world by the initial set of test cases for the lane change assistant. A detail statistic is given in Chapter 8.

The recorded *abstract situations* by the *qualitative* and *quantitative runtime monitoring* of autonomous vehicle systems during operation in the real world can be used for improvements of autonomous vehicle systems and extensions of system verification by additional simulations. This system evolution based on the runtime monitoring results enables car manufacturers to increase the safety of their autonomous vehicle systems iteratively. The following section describes the improvement of autonomous vehicle systems and the extensions of the simulations in the system verification in more detail.

7.2. System Evolution

The holistic engineering approach (cf. Fig. 4.1) envisages a iterative cycle of system development at design time and assessment during operation. Results from the runtime monitoring of autonomous vehicle systems during operation have to be transferred back into the system development in order to identify and resolve previously unknown faulty system behavior and extend the scope of the system verification by additional simulations.

The log files of recorded *abstract situations* from the runtime monitoring of autonomous vehicle systems during operation (cf. Section 7.1) have to be analyzed in order to identify the origin of faulty system behavior. The impact of the runtime monitoring results on the development of autonomous vehicle systems is two-fold:

- Abstract situations which exhibit incorrect and unsafe system behavior indicate faults in the specification, design, and implementation of the autonomous vehicle systems (cf. Section 7.2.1).

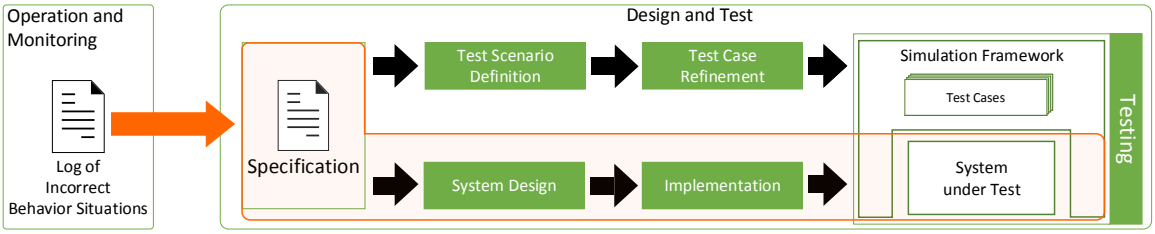


Figure 7.2.: Impact by results of qualitative runtime monitoring.

- Unverified *abstract situation* primarily indicate missing or faulty test scenarios (cf. Definition 4.5) and test cases (cf. Definition 4.6) in the test suites for the system verification of the autonomous vehicle systems (cf Section 7.2.2).

The impacts of the *qualitative* and *quantitative runtime monitoring* during operation of autonomous vehicle systems for the system evolution is described in the following sections individually.

7.2.1. Improvement by Situations with Incorrect System Behavior

Log files from the *qualitative runtime monitoring* of autonomous vehicle systems during operation in the real world contain sequences of *abstract situation* in which the autonomous vehicle systems operated incorrectly and unsafe — violating system requirements. These *abstract situations* are essential for the improvement of the autonomous vehicle systems. As shown in Fig. 7.2, the results from the *qualitative runtime monitoring* during operation of the real world primarily impact the requirements analysis, system design, and implementation of autonomous vehicle systems.
The histories of previous *abstract situation* which are recorded with each critical *abstract situation*support engineers in the manual localization, analysis, and improvement of system faults. This may include the review of requirements, system design, and implementation in order to find the origin of the faulty system behavior. Based on the identified system fault, engineers have to revise all depending development artifacts. Faults in requirements require the revision of the requirements, system design, safety analysis, and system implementation while implementation faults only influence the implementation and do not require changes of requirements, safety analysis, or system design. As results of system improvements, a version of the autonomous vehicle system is developed which is supposed to operate correctly and safely in all known *abstract situations*.
The improved versions of the autonomous vehicle systems are verified again in simulations of the system verification for existing and additionally created test scenarios and test cases. Additional test scenarios and test cases are defined based on the recorded *abstract situations* with faulty system behavior and their histories. The improved versions of the autonomous vehicle systems must operate correctly and safely in all *abstract situations* in which the previous system version exhibited an incorrect and unsafe system

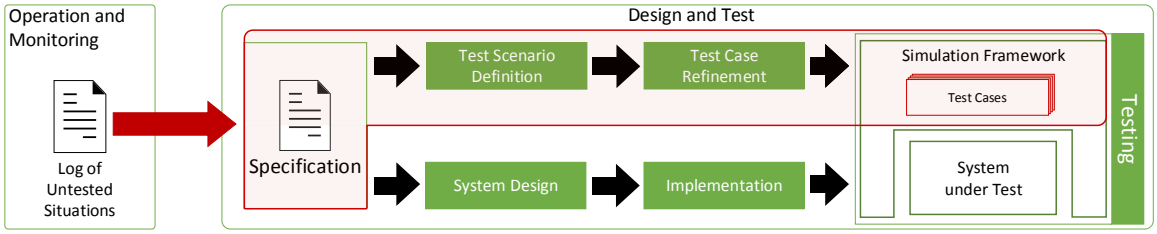


Figure 7.3.: Impact by results of quantitative runtime monitoring.

behavior. The verification for existing test scenarios and test cases ensures that no system improvements have altered the previously correct system behavior. The test scenarios and test cases which are derived from the results of the *qualitative runtime monitoring* can be added to the system test suite in addition to test scenarios and test cases from the *quantitative runtime monitoring*. For efficient verification of autonomous vehicles systems in simulation, the test suite should be a comprehensive set of diverse test cases which evaluates a large number of situations with a minimal number of test cases (cf. [Bac+17c]). Metrics for situation coverage can help to define such an efficient and diverse test suite (cf. [AHR15]).
The lane change assistant was provided for the case study as a binary software package without any source codes (cf. Chapter 8). The missing source code prohibited any improvement of the lane change assistant based on the recording of the qualitative runtime monitoring. Therefore, the case study focuses on completing the round-trip of the engineering approach by defining test scenarios and test cases from recordings of the *quantitative runtime monitoring* during operation (cf. Fig. 5.1). The next section describes the definition of test scenarios and test cases from recorded *abstract situation* by the quantitative runtime motoring in more detail.

7.2.2. Improvement by Unverified Situations

Log files from the *quantitative runtime monitoring* contain *abstract situations* from the operation of autonomous vehicle systems in the real world. These *abstract situations* have not yet been verified in simulations of the system verification. The correctness and safety of the autonomous vehicle systems in these *abstract situations* is unknown unless the *qualitative runtime monitoring* has recorded the identical *abstract situation*. *Abstract situations* from the *quantitative runtime monitoring* are not suitable for the identification, location, and correction of system faults but are beneficial for the definition of additional tests for the system verification (cf. Fig. 7.3).
For the extension of the simulation-based tests, *abstract situation* in the log files of the *qualitative* and *quantitative runtime monitoring* are transformed into *test scenarios* (cf. Definition 4.5) and *test cases* (cf. Definition 4.6). *Test scenarios* are defined from coherent traces of recorded *abstract situations* in the log files (cf. Definition 2.32) based on changes of the scenery, the behavior of the automated ego vehicle, and the behavior of all other dynamic objects in the vicinity of the automated ego vehicle. Each *test*

scenario defines a ordered trace of scenery changes and actions by dynamic objects (cf. Section 2.2.1). Changes of static objects in the scenery between two subsequent *abstract situation* are unambiguous and are directly adopted into the test scenario. The behavior of the automated ego vehicle the dynamic objects are more complex. Multiple combinations of actions by different dynamic objects can lead to identical changes of dynamic objects between two subsequent *abstract situation*. One combination of object actions is selected for each transition between subsequent *abstract situations* in the test scenarios.
Each test scenario describes the behavior of the automated ego vehicle and dynamic objects in its vicinity on an abstract level and, therefore, abstracts from concrete simulation parameters of simulation frameworks. The abstraction level of the test scenarios matches the abstraction level of the considered system requirements. For the lane change assistant, the velocities of vehicles have been recorded in relation to the automated ego vehicle, but VTD requires each vehicle's absolute velocity for the simulation.
The modeling of static and dynamic objects in *environment models* of simulations frameworks require a different level of abstraction (cf. Definition 6.2). The environment models of simulation framework describe the positioning and behavior of environment objects by the International System of Units (SI) (cf. [PM06]).
Test scenarios are parametrized by a set of test cases. The abstract description of *test scenarios* are transformed into *test cases* in order to match the level of abstraction which is required by the simulation frameworks. Each *test case* augments missing concrete parameters for the *environment model* in the simulation. The missing concrete parameters are derived from the abstract parameters in the *test scenario* by parameters fuzzing using combinatorics and parametrization algorithms, e.g., equivalence class partitioning and boundary value analysis (cf. [Ana+13; Bac+17c; Kha+17; Sip+16]). This way, a single test scenario can be verified in the simulations of the system verification for a vast number of concrete parameter constellations.
The *test cases* which have been defined from the results of the *quantitative runtime monitoring* during operation are added to the test suite of the system verification. The quantity of new test scenarios and test cases from real-world data is proportional to the amount of real-world data. The more data has been recorded, the more test scenarios and test case can be defined. Massive fleets are more likely to generate excessive real-world data and extend the scope of verified traffic situations for the autonomous vehicle systems in a shorter duration.
The *quantitative runtime monitoring* of autonomous vehicle systems during operation in the real world does not only enable the definition of additional test scenarios and test cases but also the definition of the initial test suite for the autonomous vehicle systems. Manually modeled test cases tend to represent more straightforward traffic scenarios with limited realism (cf. Chapter 8).
For the definition of an initial test suite from real-world data, the *tested situation knowledge* contains no *abstract situations*. Therefore, all encountered *abstract situations* during operation in the real world are recorded and can be used for the definition of the initial test scenarios and test cases. These test cases constitute a realistic initial test suite. The test scenarios and test cases can be reused for multiple autonomous vehicle

systems as long as the runtime monitoring framework remains consistent among these systems.
The improved versions of autonomous vehicle systems are verified in the system verification for all new and existing test cases. The new system versions must not exhibit any faulty behavior in the system verification for any of the new test cases. Otherwise, specification, design, and implementation for the autonomous vehicle systems have to be revised until all — new and existing — test cases are successfully passed. The improved version of the autonomous vehicle system is not suited for the operation in the real world until all test cases have successfully been passed.
The *quantitative runtime monitoring* of the lane change assistant during operation has been performed on recordings of test drives on public roads because a prototype vehicle has not been available for the case study (cf. Chapter 8). 1071 unknown and unverified *abstract situations* have been recorded by *quantitative runtime monitoring* for the test drive on the German highway A2. In the recording of the test drive on the German highway A39, the *quantitative runtime monitoring* recorded 959 unknown *abstract situation*.
Seven additional test scenarios and seven corresponding test cases have been modeled and verified in simulations with VTD based on the recorded unknown and unverified *abstract situation* in the recording on the German highway A2. The test scenarios and test cases were manually selected and modeled from logged traces of *abstract situations*. The coverage of the encountered *abstract situations* by all test cases — the manually modeled test cases and the seven additional realistic test cases — increased for the test drive on the German highway A2 to over 10%. A detailed statistic is given in Chapter 8.
The following section describes the definition of test scenarios and test cases from recordings of *abstract situation* by the runtime monitoring during operation in the real world in more detail.

7.2.3. Definition of Test Scenarios and Test Cases from Runtime Data

For the extension of the simulation-based tests in the system verification by realistic test cases, log files from the *qualitative* and *quantitative runtime monitoring* during operation in the real world are transformed into *test scenarios* (cf. Definition 4.5) and *test cases* (cf. Definition 4.6).
Log files contain a trace of unverified *abstract situations* and their histories of preceding *abstract situations*. The lists of unverified *abstract situations* contain *clusters* of *abstract situations* which have been subsequently recorded in cohesive time frames. Each *cluster* represents a *trace* of subsequent *abstract situation* (cf. Definition 2.32) which have not yet been considered in the simulations of the system verification (cf. Fig. 7.4).

Definition 7.1 (Cluster)**.** *A cluster in a log of the runtime monitoring framework describes a trace of subsequent* abstract situations *which have been recorded in a cohesive time frame. Any cluster corresponds to a trace of* abstract situations*.*

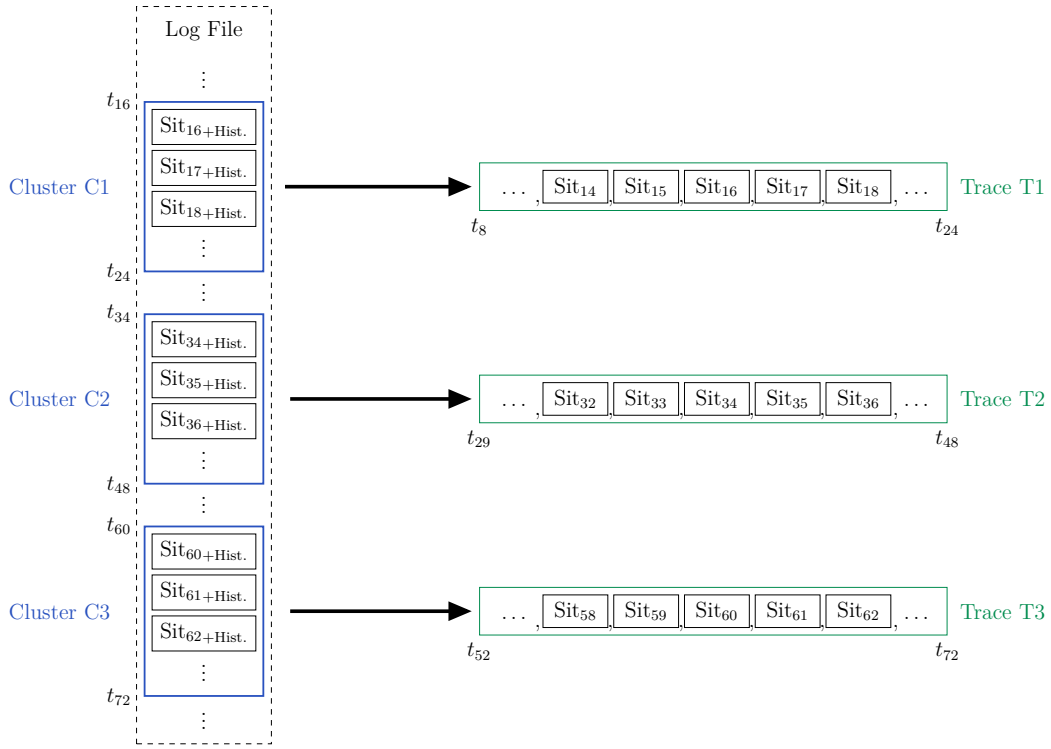


Figure 7.4.: Clusters and traces of recorded abstract situations.

Abstract situations in these *clusters* share similar histories of preceding *abstract situations*. The histories of two subsequent *abstract situation* in these *clusters* vary only in one situation because the recent unknown and unverified *abstract situation*is added to the history while the eldest entry of the history is discarded.
Cluster may even contain sequences of identical *abstract situation*. A situation may occur for multiple processing cycles and, therefore, is recorded multiple times. For the efficiency of the simulation in the system verification, any subsequent duplicates of an *abstract situation* are discarded in the aggregation. The timing of environment activities is not considered for test scenarios.
In the rare case that a *cluster* only contains a single unknown and unverified situation. The related history of preceding *abstract situations* should contain enough *abstract situations* to model a sufficient input trace for the definition of test scenarios and corresponding test cases. An exception would be the start of the runtime monitoring during operation. The encountered *abstract situations* at the beginning of the runtime monitoring might not sufficiently fill the history in order to model valuable test scenarios.
The definition of test scenarios and test cases from *traces* of recorded *abstract situations* is divided into three tasks:

1. The identification and definition of transitions from analysis of possible actions of dynamic objects between subsequent *abstract situation* (cf. Section 7.2.3.1).

2. The slicing of test scenarios (cf. Section 7.2.3.2).

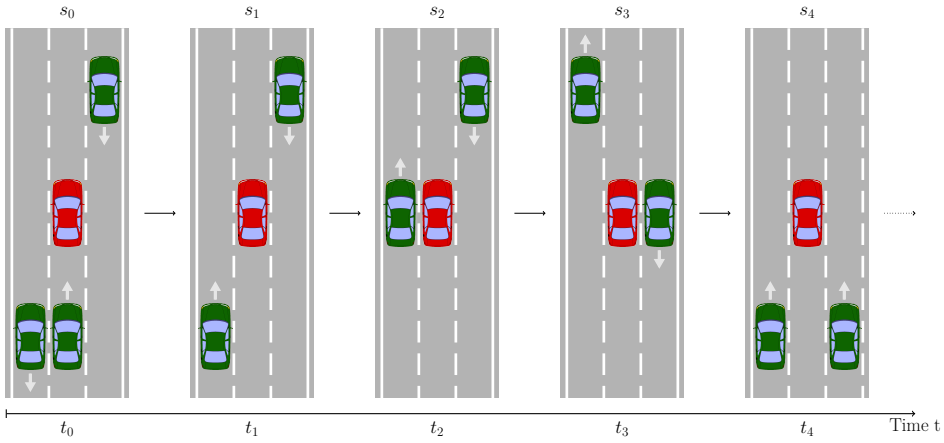


Figure 7.5.: Example trace of recorded traffic situations.

3. The parametrization of test scenarios as test cases by deriving parameters of the environment models from parameters of *abstract situation* in the test scenarios (cf. Section 7.2.3.3).

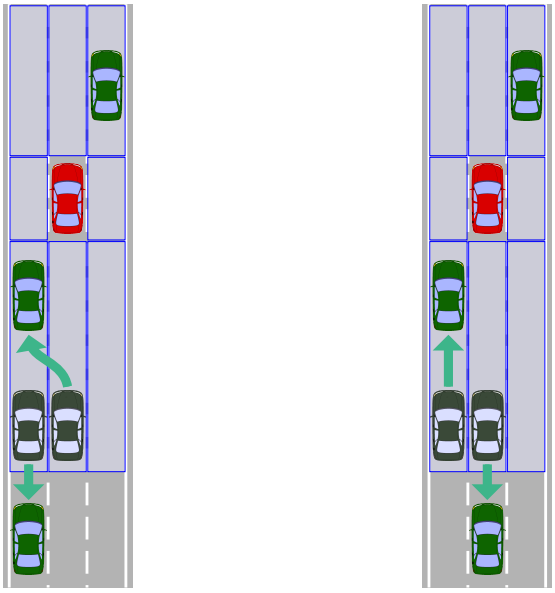
The following sections describes each task in more detail.

7.2.3.1. Identification of Situation Transitions

Abstract situations solely describe the states of the autonomous vehicle systems and their environments at certain timestamps on the abstraction level of the system requirements (cf. Fig. 7.5). However, simulations frameworks require the definition of actions for all dynamic objects in environment models. The automatic randomization of roads, scenery, and traffic in simulations is no solution for the remodeling of recorded *abstract situation* traces by the runtime monitoring framework (cf. Chapter 8). For the definition of *test scenarios* from recorded traces of *abstract situations*, the actions of dynamic objects (cf. Definition 2.29) and the changes of the scenery — the road and other static objects — (cf. Definition 2.27) between all subsequent *abstract situation* of the traces have to be identified. The following sections provide more detail descriptions about the identification of changes and dynamic objects and the scenery.

7.2.3.1.1. Changes of the Scenery

The changes in the scenery between subsequent *abstract situations* can be defined based on the changes of individual *situation parameters* for its static objects. Situation parameters corresponds to the types in the *abstract representation* of the situation (cf. Definition 4.4) in the runtime monitoring framework. The changes of static objects entirely define the changes in the scenery between subsequent *abstract situation*. Unlike dynamic objects (cf Section 7.2.3.1.2), combinations of different scenery changes have not to be considered because static objects are not able to switch their positions with other dynamic or static objects. Otherwise static objects would be dynamic objects.



(a) One vehicle leaves the zone while another vehicle changes into the zone.

(b) Vehicle remains in zone with different velocity while an other vehicle leaves the zone.

Figure 7.6.: Possible object maneuvers for changes between situations s_0 and s_1.

Definition 7.2 (Situation Parameter)**.** *A situation parameter corresponds to one specific type in the type hierarchy for the the* abstract representation *of traffic situations and system states (cf. Definition 4.4) in the runtime monitoring framework.*

Changes of static objects, e.g., roads, their lanes, their markings, house, and trees, can be individually assessed and documented. For example, road markings can either remain solid or dashed or change from solid to dashed or vice versa between two subsequent *abstract situations*. Static objects and their changes are later considered in the definition of the test cases (cf. Section 7.2.3.3).

7.2.3.1.2. Behavior of Dynamic Objects

The identification of actions by dynamic objects between two subsequent *abstract situation* is more complicated than changes in the scenery. The number of combinations of actions for the dynamic objects in the initial *abstract situation* leading to subsequent *abstract situation* can be vast (cf. Fig. 7.6).
While the vehicles in the top right zones in Fig. 7.6a and Fig. 7.6b remain in the same zones over both *abstract situations*, the positioning of the lower two vehicles offers multiple possibilities for combinations of maneuver by these two vehicles. The initial positions of the vehicles behind the automated ego vehicle is displayed in Fig. 7.6a and Fig. 7.6b by the dark green car while the final position is displayed by the light green car.

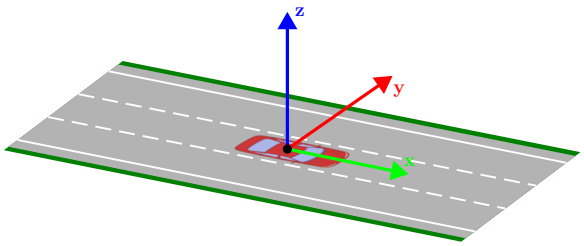


Figure 7.7.: 3D coordinate system for the movements of dynamic objects on roads.

In Fig. 7.6a, the vehicle initially in the left rear falls out of the zone while rear vehicle in the mid-rear zone changes to the left neighbor lane.
In Fig. 7.6b, the vehicle behind the automated ego vehicle on the left lane accelerates and remains in the zone while the vehicle behind the automated ego vehicle decelerates and falls out of the zone.
In both examples, the final *abstract situation* has one vehicle on the right lane in front of the automated ego vehicle and one vehicle on the left neighbor lane behind the automated ego vehicle. For useful simulations, all possible combinations of actions by different dynamic objects should be considered in the definition of test scenarios.
For each type of dynamic objects in abstraction situation (cf. Fig. A.2), the set of possible action has to be defined. In the 3D space, the ISO 8855 (cf. [Int11a]) defines the x-axis in driving direction of the vehicle, the y-axis perpendicular to the x-axis pointing to the left, and the z-axis pointing upwards (cf. Fig. 7.7). The position and orientation of objects can be changed in translational and rotational directions for each of the three axes (cf. [Lee82]). These possibilities will result in 12 possible movements if the negative and positive movement for translations and rotations are considered separately.
Traffic is commonly analyses and simulated in the 2D space with road being the base plane (cf. Section 2.2.1.1.2). Traffic objects, e.g., vehicles, move only in the 2D space of the road (cf. [Bar10]). Translations in the vertical direction of z-axis are commonly not possible in this view and, therefore, are not considered by autonomous vehicle systems. The restriction of movement in the direction of the z-axis reduces the number of possible movements to 10. In case, the slope of roads and vehicle traction control systems are not regarded, the number of possible movement can be reduced to 6 because any rotations around the plane axis — x-axis and y-axis — can be omitted (cf. Fig. 7.7). The view of traffic in the 2D plane is the lowest possible level of abstraction. Actions of dynamic objects can be described at least described by translations and rotations in the 2D plane. For most use cases, it is beneficial to describe actions by dynamic objects on a high level of abstraction. For vehicles, Nagel et al. have defined in [NE91] a set of 17 maneuvers. In this thesis, the term *maneuver* are generalized to all dynamic objects. Therefore, A maneuver is any action performed by any dynamic object. Therefore, the terms *action* and *maneuver* are used interchangeably in this thesis. Bajcsy et al. extended this list by the maneuver *standing* (cf. [BN96]). The 18 maneuvers are

1. start and continue,
2. follow lane,
3. cross intersection,
4. merge to the left/right lane,
5. u-turn to the left/right,
6. slowdown to right road edge and stop,
7. back up,
8. turn left/right,
9. reverse direction,
10. approach obstacle ahead,
11. overtake,
12. stop in front of an obstacle,
13. pass obstacle to the left/right,
14. start after preceding car,
15. follow a preceding car,
16. enter parking slot,
17. leave parking slot, and
18. standing.

Tölle later reduced in [Töl96] the number from 17 maneuvers to 9 maneuvers. The maneuvers can be used as basic maneuverer set for the identification of vehicles actions between subsequent *abstract situations*. Maneuvers can be further characterized by additional parameters, e.g., velocity or acceleration, or combined to form complex maneuvers. For other types of dynamic objects, similar sets of basic actions (maneuvers) have to be defined.

Definition 7.3 (Maneuver)**.** *A maneuver is any action performed by any dynamic objects, e.g., vehicles. A maneuver can be instantaneous or have some arbitrary duration. If not defined otherwise, maneuvers can be combined to form more complex maneuvers.*

For the engineering approach, maneuvers by dynamic objects have to interpret over the *abstract situations*. Each basic maneuver corresponds to specific value changes by a fixed set of parameters in the *abstract representation*, e.g., the position of the dynamic objects (cf. Fig. A.2). Therefore, the maneuvers of dynamic objects for test scenarios are derived from the set of *situation parameter* changes between two *abstract situation* (cf. Definition 7.2). All identified maneuvers between *abstract situations* have to account precisely for all changes of relevant situation parameters between these *abstract situations*. For the lane change assistant, the basic maneuvers of dynamic objects have been interpreted based on the relative positioning of these objects in relation to the automated ego vehicle and their position on lanes. The positioning by lane and relative distance enables the categorization of the automated vehicle's vicinity into *zones* (cf. Fig. 6.11). Possible maneuvers of dynamic objects in the abstract representations of situations for the lane change assistant are:

1. drive into zone,
2. fall out of zone,
3. change into zone from left / right,
4. leave zone to the left / right,

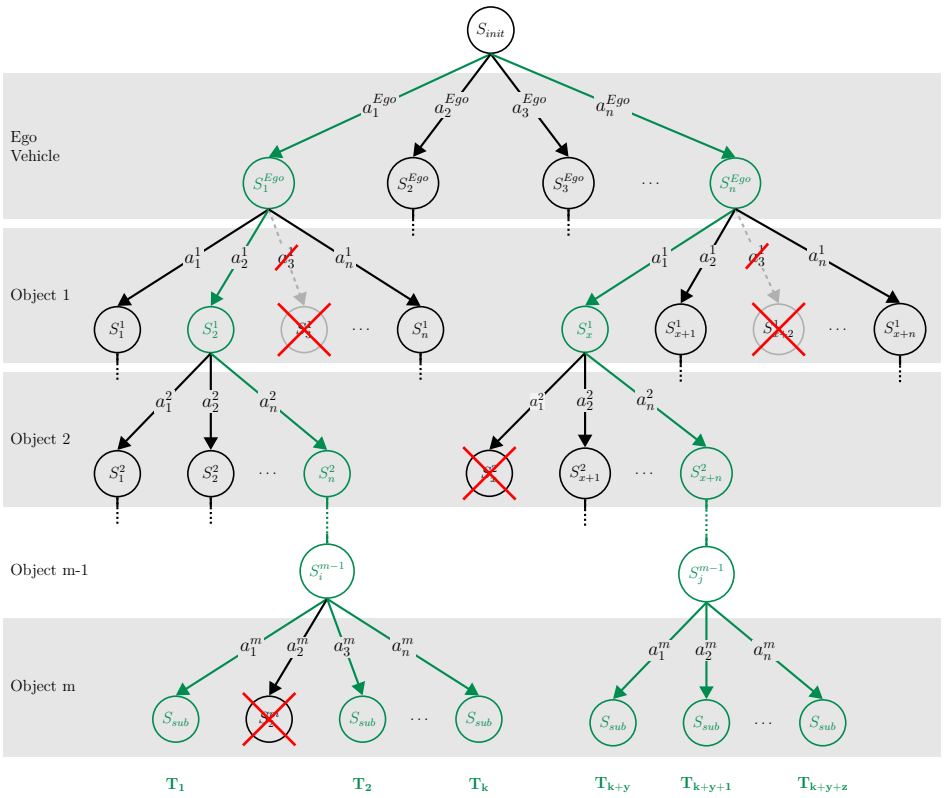


Figure 7.8.: Maneuver tree for the identification of transitions between subsequent abstract situations.

5. change to front zone,
6. change to rear zone,
7. change to left zone,
8. change to right zone

The interpretation of object maneuvers over parameters of *abstract situations* enables the identification of transitions between subsequent *abstract situations*. Transitions between subsequent *abstract situations* correspond to valid combinations of object maneuvers between subsequent *abstract situations*. Valid combinations of object maneuvers are processed from the initial *abstract situation* S_{init} by iteratively applying maneuvers to the dynamic objects of the *abstract situation* S_{init} and comparing the resulting *abstract situations* to the original subsequent *abstract situation* S^{sub} from the situation trace.
As shown in Fig. 7.8, the processing of maneuver combinations results in a *maneuver tree*. For all dynamic objects $\mathbf{O}$ in the initial *abstract situation* S_{init} — including the automated vehicle itself — all possible maneuvers $a_i^{o_k} \in A^{o_k}$ are successively applied to the dynamic object o_k in the initial *abstract situation* S_{init}. The application of object maneuvers $a_i^{o_k} \in A^{o_k}$ to the initial *abstract situation* o_k result each in an intermediate *abstract situation* S_j^k.
Maneuvers by dynamic objects may be restricted by static objects of the scenery, e.g., the road, its lanes, and their markings. For example, a solid lane marking between two adjacent lanes prohibits any lane change by any vehicle between these two lanes.

These maneuvers must be excluded from the analysis (cf. maneuver $a_3^{o_1}$ for object 1 in Fig. 7.8). Each valid maneuver $a_i^{o_k}$ for a dynamic object o_k results in an intermediate abstract representation $S_i^{o_k}$. An intermediate *abstract situation* will be valid if it partially matches the original subsequent *abstract situation* S_{sub}. Otherwise the intermediate *abstract situation* is discarded (cf. *abstract situation* S_x^2 and S_2^m in Fig. 7.8).
Maneuvers $a_i^{o_{k+1}} \in A^{o_{k+1}}$ of the remaining dynamic objects in the initial situation $o_{k+1} \in \mathbf{O}$ are subsequently applied to valid intermediate *abstract situation* S_j^k from the maneuvers of dynamic object o_k. After all dynamic objects have been processed *abstract situation* S_i^m remain which must match the targeted subsequent *abstract situation* S^{sub}.
The automated ego vehicle (*Ego Vehicle*) is supposed to be the first analyzed object in the identification of combinations of object maneuvers because its maneuver may impact the maneuvers of all dynamic objects. In the ego-centric view of the *abstract situation*, maneuvers by the automated ego vehicle are not directly apparent because the automated ego vehicle always remains at the same position.
Maneuvers of the automated ego vehicle are identified based on joint actions by all static and dynamic objects in its vicinity. As shown in Fig. 7.9, the action by the automated vehicle — a lane change in positive direction of the y-axis (left) — is represent in the *abstract situation* by inverted movements of all other objects in negative direction of the y-axis (right) (cf. Fig. 7.7).
As for situation s_6 in Fig. 7.9, new vehicles may be introduced (cf. rear right green car) and lane properties e.g., markings change. Lanes in the *abstract situation* will be omitted if the automated ego vehicle changes to the rightmost resp. leftmost lane. The omitted lanes have the type *non-existing* (cf. Fig. 3.9). In the abstract representation of the runtime monitoring, the automated vehicle can perform the following maneuvers:

- accelerate,
- decelerate,
- change to the left neighbor lane, and
- change to the right neighbor lane.

Any maneuverer by the automated ego vehicle has a significant impact on the position of static and dynamic objects in its vicinity. After the maneuver by the automated ego vehicle, the actions of dynamic objects in the initial *abstract situation* S_{init} are successively evaluated.

7.2.3.1.3. Introduction of Ghost Objects

For some situation transitions, it might be necessary to consider dynamic objects which are not contained in the initial *abstract situation* S_{init} but could drive into the range of the sensor perception for the subsequent *abstract situation* S_{sub}. These dynamic objects have to be considered as so-called *ghost objects* in the identification of maneuver combinations for each type of dynamic objects as in the data structure of the *abstract situation* (cf. Definition 7.4).

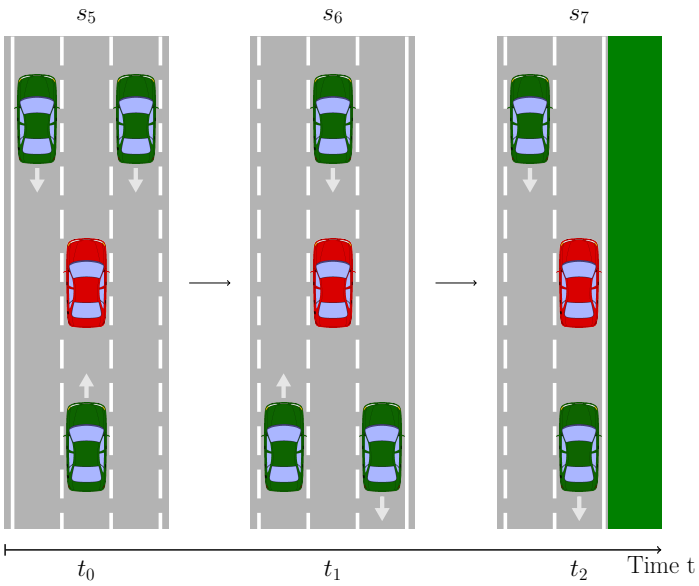
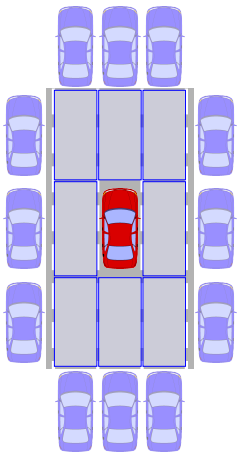


Figure 7.9.: Identification of actions for the automated ego vehicle.

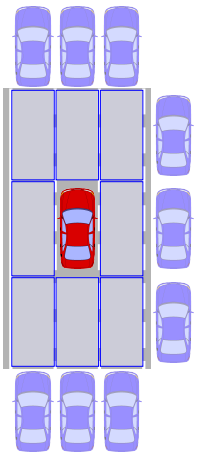
Definition 7.4 (Ghost Object)**.** *A ghost object represents a dynamic object of a specific type which maneuvers would allow the object to appear in the subsequent* abstract situation *though the object has not been present in the initial* abstract situation.

As shown in Fig. 7.10a, *ghost objects* are virtual dynamic objects which are positioned around the boundary of the *abstract situation*. *Ghost objects* are defined based on relevant parameters of the *abstract situation*. The position of *ghost objects* are determined for each type of dynamic objects based on the inversion of their maneuvers resp. the reversion of the corresponding parameter changes in the abstract representation of situations from a valid position for objects of this type within the *abstract situations*. For example, the position for a *ghost vehicle* would be determined by applying *a lane change to the right lane* to the position of vehicles in the *front right zone*. The lane change to the right lane is the inversion of a *lane change to the left lane* for the *ghost object* positioned adjacent right to the *front right zone* (cf. Fig. 7.10a). The inversion of maneuvers is trivial in most cases as long as the level of abstract is sufficient. The directional maneuvers of vehicles can all be inverted by inverting their changes of situation parameters. For the lane change assistant, the list of maneuvers for a dynamic object already contains all pairs of original and inverse maneuvers — lane change to the left/right and accelerate/decelerate. *Ghost objects* are only considered at positions which reside outside the boundary of the *abstract situation*. It is possible that multiple inverted maneuvers applied to a different valid position within the *abstract situations* result in the identical position for a `ghost object`.

The positioning of *ghost objects* can be restricted by situation parameters and static object of the *abstract situation*. Inverse maneuvers must consider the restrictions of their



(a) General positioning of ghost objects.



(b) Restriction on ghost objects.

Figure 7.10.: Ghost objects for determination of objects actions.

original maneuvers. Any inverse maneuver which violates a restriction of the original maneuvers in its application to a position within the *abstract situation* must be excluded from the positioning of *ghost objects*. As shown in Fig. 7.10b, a solid lane marking on the left neighbor lane would prohibit any lane changes from *ghost objects* adjacent to the left of this left neighbor lane. For the lane change assistant, *ghost objects* are restricted based on the type of the lane marking (solid marking) and the lane itself (*non-existing* lane). Paths from the root to valid leafs of the *maneuver tree* represent accurate combinations of object maneuvers between the two subsequent *abstract situation* S_{init} and S_{sub} from the input trace. Valid combinations of object maneuvers for the maneuver tree in Fig. 7.8 are e.g., Eq. (7.1) and Eq. (7.2). Each path denotes a valid transition between the subsequent *abstract situation* S_{init} and S_{sub}.

$$\mathbf{T_1} : S_{init} \xrightarrow{a_1^{Ego}} S_1^{Ego} \xrightarrow{a_2^1} S_2^1 \xrightarrow{a_n^2} S_n^2 \dashrightarrow S_i^{m-1} \xrightarrow{a_1^m} S_{sub} \tag{7.1}$$

$$\mathbf{T_{k+y+z}} : S_{init} \xrightarrow{a_1^{Ego}} S_1^{Ego} \xrightarrow{a_2^1} S_2^1 \xrightarrow{a_n^2} S_n^2 \dashrightarrow S_j^{m-1} \xrightarrow{a_n^m} S_{sub} \tag{7.2}$$

7.2.3.1.4. Integration as Situation Graphs

For the definition of test scenarios, input traces of *abstract situation* and all identified transitions between two subsequent *abstract situation* of these traces can be integrated into a *situation graph* (cf. Definition 7.5). As shown in Fig. 7.11, the *situation graph* is a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ where the set of nodes $\mathbf{V}$ denotes the *abstract situation* and the set of edges $\mathbf{E}$ corresponds to the identified transitions between two *abstract situation*. Abstract situations occurring in multiple input traces are represented by a single node in the *situation graph*. The input trace of *abstract situation* In_i for the extract of a *situation graph* by Fig. 7.11 are defined by Eqs. (7.3) to (7.6).

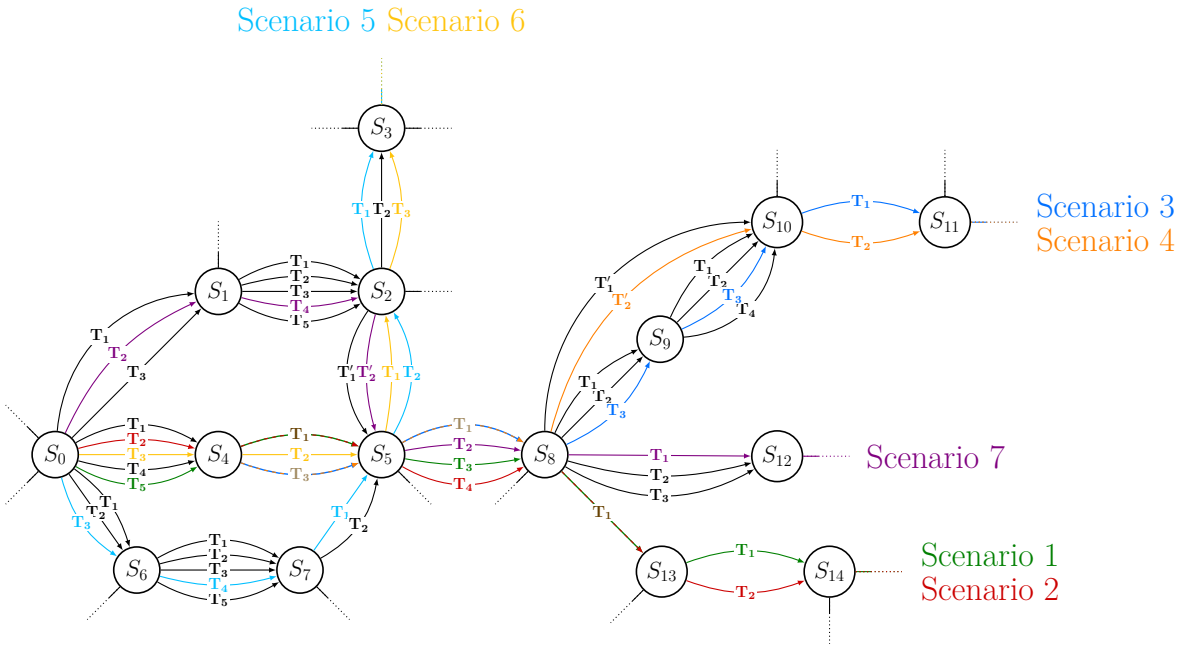


Figure 7.11.: A *situation graph* for the definition of test scenarios.

Definition 7.5 (Situation Graph)**.** *A situation graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ *represents traces of* abstract situation *with the identified transitions between two* abstract situation*. The set of nodes* $\mathbf{V}$ *represents the set of* abstract situations*, and the set of edges* $\mathbf{E}$ *corresponds to the transitions between two* abstract situations*.*

$$In_1 : S_0 \rightarrow S_4 \rightarrow S_5 \rightarrow S_8 \rightarrow S_{13} \rightarrow S_{14} \rightarrow \dots \tag{7.3}$$
$$In_2 : S_0 \rightarrow S_4 \rightarrow S_5 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_{11} \rightarrow \dots \tag{7.4}$$
$$In_3 : S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_5 \rightarrow S_8 \rightarrow S_{12} \rightarrow \dots \tag{7.5}$$
$$In_4 : S_0 \rightarrow S_6 \rightarrow S_7 \rightarrow S_5 \rightarrow S_2 \rightarrow S_3 \rightarrow \dots \tag{7.6}$$

The *situation graph* may contain cycles because the sequence in which situations may occur in the real world is arbitrary. Even *self-cycles* from and to the same *abstract situation*are possible. Unless time or subsequent occurrences of a situation are essential for the environment modeling in simulations, the *situation graph* should be free of *self-cycles*. Multiple subsequent occurrences of identical *abstract situation* are eliminated in the prepossessing of the input trace for the dentition of the test cases.
Transitions between *abstract situation* in the *situation graph* are directed edges. The maneuvers of dynamic objects in a transition $\mathbf{T_i}$ are only valid for the given direction between two *abstract situation* S_i and S_{i+1}. An inversion does not exist for all transition $\mathbf{T_i}$. For example, a lane marking might allow lane changes in one driving direction but does not allow lane changes in the opposite driving direction.
A possible inversion of any transition should only be included in the *situation graph* if there has been an input trace with reversed order of *abstract situations* and the inversion

of object maneuvers has been identified in the corresponding analysis. This practice enables the *situation graph* to record the order in which *abstract situations* have occurred in the real world and use this knowledge for further analysis.
Test scenarios are defined based on paths through the *situation graph* by selecting one transition $\mathbf{T_i}$ for each pair of subsequent *abstract situation* S_i and S_{i+1}. Even though, cycles in the *situation graph* allow to model infinite test scenarios, defined test scenarios should be finite in order provide result in reasonable time. For the *situation graph* Fig. 7.11, seven test scenarios have been exemplary defined.
The test scenarios 1 and 2 represent two test scenarios for the same input trace of *abstract situations* In_1. The sequence of *abstract situations* of both test scenarios are identical, but the transitions between two subsequent *abstract situations* vary.

$$\text{Scenario 1} : S_0 \xrightarrow{\mathbf{T_5}} S_4 \xrightarrow{\mathbf{T_1}} S_5 \xrightarrow{\mathbf{T_3}} S_8 \xrightarrow{\mathbf{T_1}} S_{13} \xrightarrow{\mathbf{T_1}} S_{14} \rightarrow \dots \tag{7.7}$$

$$\text{Scenario 2} : S_0 \xrightarrow{\mathbf{T_2}} S_4 \xrightarrow{\mathbf{T_1}} S_5 \xrightarrow{\mathbf{T_4}} S_8 \xrightarrow{\mathbf{T_1}} S_{13} \xrightarrow{\mathbf{T_2}} S_{14} \rightarrow \dots \tag{7.8}$$

Scenario 3 and scenario 4 are two test scenarios defined for the input trace of *abstract situation* In_2 (cf. Eq. (7.9)). These two test scenarios differentiate themselves by transitions between S_8 and S_{10}. The test scenario 4 models a transition from S_8 directly to S_{10} while scenario 3 models an additional *abstract situation* S_9 and corresponding transitions. However, the simulation of scenario 4 will produce a trace of *abstract situation* similar to scenario 3 — including *abstract situation* S_9. The absence of *abstract situation* S_9 in test scenario 4 may originate from a short appearance of *abstract situation* S_9 in the real world within two processing cycle of the runtime monitoring framework.

$$\text{Scenario 3} : S_4 \xrightarrow{\mathbf{T_3}} S_5 \xrightarrow{\mathbf{T_1}} S_8 \xrightarrow{\mathbf{T_3}} S_9 \xrightarrow{\mathbf{T_3}} S_{10} \xrightarrow{\mathbf{T_1}} S_{11} \rightarrow \dots \tag{7.9}$$

$$\text{Scenario 4} : S_4 \xrightarrow{\mathbf{T_3}} S_5 \xrightarrow{\mathbf{T_1}} S_8 \xrightarrow{\mathbf{T'_2}} S_{10} \xrightarrow{\mathbf{T_2}} S_{11} \rightarrow \dots \tag{7.10}$$

Scenario 5 models a test scenario for the input trace In_4 (cf. Eq. (7.11)). The input trace In_4 shares the *abstract situation* S_5 with input traces In_1 and In_2.
Intersections and overlapping of traces enable to combine sub-paths from different input traces into a novel test scenario. The test scenario 6 consists of the sub-path $S_0 \xrightarrow{\mathbf{T_3}} S_6 \xrightarrow{\mathbf{T_4}} S_7 \xrightarrow{\mathbf{T_1}} S_5$ from input traces In_1 resp. In_2 and the sub-path $S_5 \xrightarrow{\mathbf{T_2}} S_2 \xrightarrow{\mathbf{T_1}} S_3$ from the input trace In_4 (cf. Eq. (7.12)).

$$\text{Scenario 5} : S_0 \xrightarrow{\mathbf{T_3}} S_6 \xrightarrow{\mathbf{T_4}} S_7 \xrightarrow{\mathbf{T_1}} S_5 \xrightarrow{\mathbf{T_2}} S_2 \xrightarrow{\mathbf{T_1}} S_3 \rightarrow \dots \tag{7.11}$$

$$\text{Scenario 6} : S_0 \xrightarrow{\mathbf{T_3}} S_4 \xrightarrow{\mathbf{T_2}} S_5 \xrightarrow{\mathbf{T_1}} S_2 \xrightarrow{\mathbf{T_2}} S_3 \rightarrow \dots \tag{7.12}$$

The test scenario 7 models the possibility of transitions in different directions between two *abstract situation* (cf. Eq. (7.13)). While input trace In_4 contains a transition from *abstract situation* S_5 to *abstract situation* S_2, the input trace In_3 includes a transition in reverse direction — from *abstract situation* S_2 to *abstract situation* S_5. Scenario 7 defines a test scenario for this reversed direction between *abstract situation* S_2 and S_5.

$$\text{Scenario } 7 : S_0 \xrightarrow{\mathbf{T_2}} S_1 \xrightarrow{\mathbf{T_4}} S_2 \xrightarrow{\mathbf{T'_2}} S_5 \xrightarrow{\mathbf{T_2}} S_8 \xrightarrow{\mathbf{T_1}} S_{12} \rightarrow \ldots \tag{7.13}$$

For the lane change assistant, seven test scenarios have been manually defined from recordings of the runtime monitoring in test drives on German highways A2 (cf. Appendix A.3). Seven coherent sequences of *abstract situations* have been selected from its log file. One test scenario has been defined for each sequence. The sets of object maneuvers and scenery changes between *abstract situations* of these sequences have been manually defined. A *maneuver tree* or a *situation graph* have not been used for the identification of transitions between subsequent *abstract situations*. The test scenarios have been further parametrized by test cases for simulations in VTD (cf. Section 7.2.3.3). Even though all recorded situation traces by the runtime monitoring of autonomous vehicle systems during operation in the real world should be integrated into a *situation graph*, not all possible test scenarios within the *situation graph* should be used in the system verification of the autonomous vehicle systems. A large number of test scenarios would have little to no impact for the system verification because they would contain scenarios and situations which have already been considered by other test scenarios.
As indicated by Fig. 7.11, *situation graphs* may result in long and complex test scenarios. With the increasing complexity of test scenarios, it becomes more difficult to precisely define corresponding *environment models* for simulations. The following section introduces the *slicing* of test scenarios in order to limit the content of test scenarios to relevant changes in the scenery and core movements of dynamic objects.

7.2.3.2. Slicing of Test Scenarios

Test scenarios identified in the *situation graph* tend to be very long — if not infinite — and contain a vast number of vehicle maneuvers and environment changes. Long test scenarios are difficult to model in simulation frameworks, e.g., VTD. Maneuvers by dynamic objects and scenery changes in environment models have to be correctly aligned in the environment models in order for simulations to correctly represent these test scenarios.
This alignment becomes more difficult with increasing duration of test scenarios because the likelihood for deviating behavior of the automated ego vehicle and other dynamic objects increases with the duration of the simulations resp. test scenarios. The complexity of dependencies between dynamic objects themselves and the automated ego vehicle increases with the length of test scenarios. This complexity increases the difficulty to cause the envisaged behavior of the autonomous vehicle systems in later stages of the test scenarios. Small modeling faults in earlier stages of test scenarios may significantly impact later stages of the test scenarios. For example, a vehicle which has previously change the lane and has been overtaken by the automated ego vehicle might not accelerate in time to overtake the automated ego vehicle in later stages of the test scenarios. Incorrectly aligned simulations will not represent their corresponding test scenarios correctly.
Slicing of test scenarios which have been identified in the *situation graph* is introduced in order to mitigate the difficulties in the alignment of scenery changes and maneuvers by

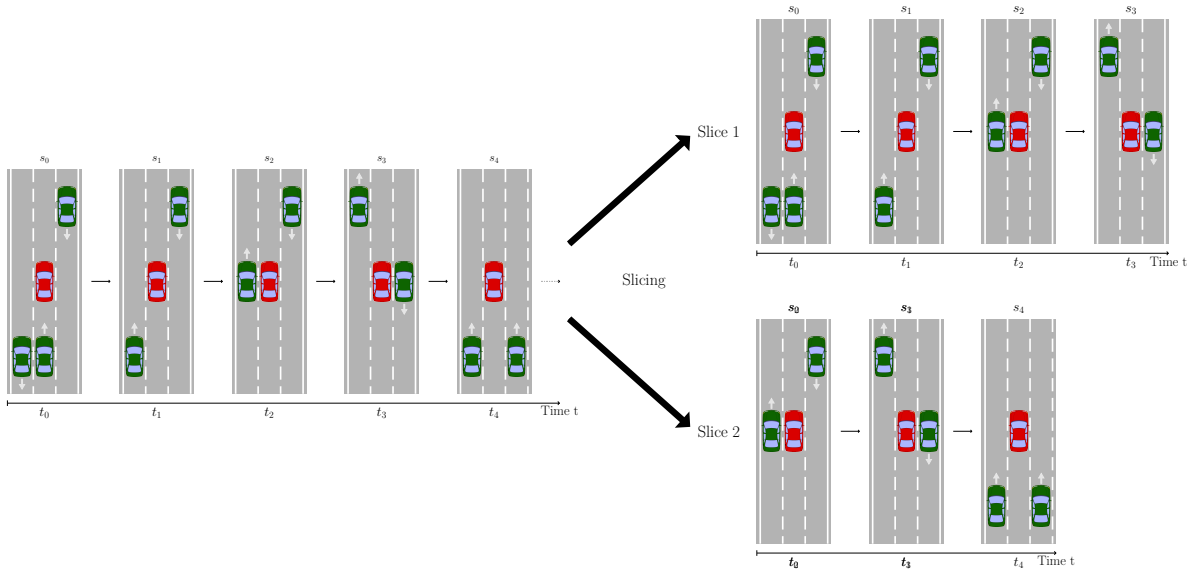


Figure 7.12.: Slicing of traffic situation traces.

dynamic objects in long test scenarios. As shown in Fig. 7.12, long test scenarios are *sliced* in to a set of multiple shorter test scenarios. These short test scenarios have a limited focus. For example, a scenario could evaluate the lane changing by the automated ego vehicle while another scenario evaluates the subsequent driving behind a preceding vehicle. The slices substitute the initial long test scenarios and represent the new test scenarios. The behaviors of dynamic objects in the simulations of these short test scenarios are less likely to diverge from the intended behavior for these objects by the test scenarios than for the extended test scenario.

Definition 7.6 (Slicing)**.** *The partition of long test scenarios into specialized sub-scenarios with smaller durations.*

Different strategies can be applied for the *slicing* of test scenarios. All *slicing strategies* have to result in slices S with sufficient *prefixes* S_{pre} and *suffixes* S_{suff} in addition to the critical behavior of the autonomous vehicle systems.
Prefixes S_{pre} are sub-scenarios at the beginning of a slice $S = S_{pre}S'$ where S' denotes the remain part of the slice. There exist no situations or object maneuvers before a *prefix* in a slice. *Prefixes* are required in order to analyze the behavior of the autonomous vehicle systems and all objects in its environment that lead to emerging critical action by the autonomous vehicle systems. Otherwise, the reason for a critical system action cannot be identified because the test scenarios directly start with the critical action for the autonomous vehicle system in the first *abstract situation*.
Suffixes S_{suff} denotes sub-scenarios at the end of a slice $S = S'S_{suff}$ with no situation or object maneuver existing in the slices after the *suffixes*. *Suffixes* are necessary for the evaluation of system behavior after the occurrence of a potential critical action by the autonomous vehicle systems. A sufficient long suffix of situations has to pass after any

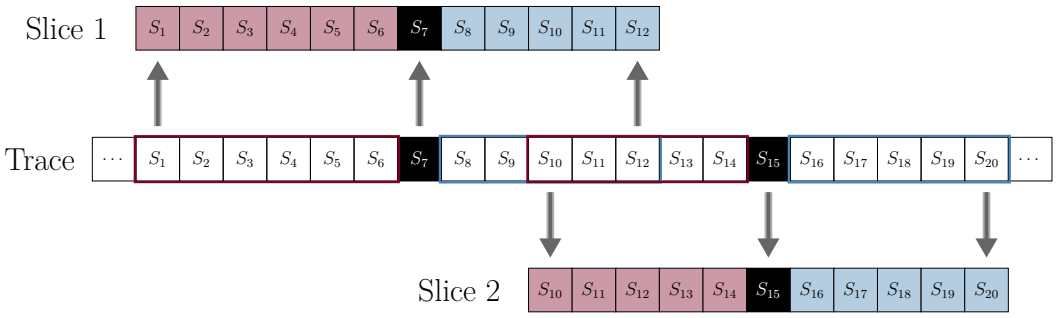


Figure 7.13.: Prefixes and suffixes in slicing of situation traces.

maneuver by the automated vehicle or any dynamic object in order to soundly evaluate if the autonomous vehicle systems have reached a safe system state.

One slicing strategy is to use the set of maneuvers for the automated ego vehicle as a reference (cf. Section 7.2.3.1). Long test scenarios can be sliced based on the maneuvers by the automated ego vehicle, e.g., lane changing, overtaking, or decelerating. The length of corresponding slices is determined based on *prefixes* and *suffixes* of similar maneuver instances in other test scenarios. The approach learns the necessary length of *prefixes* and *suffixes* for the sound evaluation of each maneuver in the simulation of the slices. The conservative approach is to use the maximum *prefix* and maximum *suffix* for each maneuver by the autonomous vehicle in the slicing of test scenarios.

The slicing strategy based on maneuvers of the automated vehicle requires the presence of a sufficient extensive *situation graph* for the identification of *prefixes* and *suffixes*. Multiple slices can be identified for a single long test scenario from the *situations graph* (cf. Fig. 7.12). These slices will include identical situations and maneuvers by dynamic objects if the *prefixes* and *suffixes* of slices overlap within the long test scenario (cf. Fig. 7.13). In Fig. 7.13 the *black* situation contain the ego maneuvers of the automated ego vehicle The corresponding prefixes and suffices for each ego meanuver lead the corresponding slices. Other slicing strategies are possible. For example, data analysis techniques, e.g., Sequential Pattern Mining (cf. [AS95; ME10; Szl18]), could be incorporated.

The slicing of long test scenarios from the *situation graph* simplifies the evaluation of long test scenarios by identifying for new test scenarios from short slices. In general, short test scenarios are evaluated more efficiently than the initial long test scenarios. Especially for manual valuation, test engineers do not have to assess long stretches of unimportant system behavior and environment changes until the interesting critical system behavior emerges. Short test scenarios directly lead to critical system behavior. In case of faulty system behavior, short test scenarios take less time to reevaluate the improved autonomous vehicle systems.

For a comprehensive verification of autonomous vehicle systems also complex scenarios with multiple maneuvers by the automated ego vehicle have to be evaluated. The short test scenarios (slices) can be combined to form longer, unknown and more complex test scenarios in order to evaluate the behavior of the autonomous vehicle system for an unknown sequence of maneuvers. Opposed to the long scenarios directly from the

situation graph, short test scenarios offer the possibility to combine test scenarios to test scenarios which have not yet been considered in the *situation graph*.
As a result of the *slicing*, a set of short test scenarios for the system verification is determined. These short test scenarios describe the state and changes of the scenarios, maneuvers of the automated ego vehicle and dynamic objects in its vicinity on an abstract level of detail. However, the level detail required by simulation frameworks may be more detailed — requiring data which is not defined by the parameters of the test scenarios. Therefore, the test scenarios are parametrized as test cases for their usage in the simulation frameworks. This parametrization by test cases is described in the following section.

7.2.3.3. Parametrization by Test Cases

Test scenarios describe changes in the scenery and the behavior of dynamic objects on an abstract level and abstract from physical parameters of the real world. The abstraction level of test scenarios is inherited from the abstraction of the runtime monitoring framework. The value ranges for these abstract parameters represents a classification of real-world parameters concerning predefined constraints of the requirements (cf. Section 6.2.2.2.1).
For example, the runtime monitoring of the lane change assistant only recorded relative velocities between dynamic objects and the automated ego vehicle. The possible values for the relative velocity are defined in comparison to a predefined limit. The absolute velocities of each dynamic objects are not recorded by the runtime monitoring framework and resulting test scenarios for the lane change assistant.
Simulation frameworks require a more detailed description of the environment. Environment models describe the scenery and dynamic objects on a level of abstraction which is similar to the physical parameters of the real world (cf. Definition 6.2). In these environment models, static and dynamic objects are initially placed in a virtual world at specific positions and with specific orientations.
Environment models have to sufficiently model the scenery with roads and relevant static objects present in the test scenarios. Roads in these environment models have to match the road layout with its number of lanes and types of road markings in the test scenarios. Roads are the foundation for the placement and orientation of all static and dynamic objects in the environment models — including the automated ego vehicle. Roads are complemented by relevant static objects from the test scenarios, e.g., parking cars or construction sides. While dynamic objects change their positions over time, static objects remain static at their global position in the virtual world. However, their position may change over time in relation to the behavior of the automated ego vehicle.
For dynamic objects in test scenarios, e.g., other vehicles, their behavior over time has to be specified by corresponding environment models. Time-dependent parameters specify the behavior of dynamic objects. For example, VTD requires the absolute velocity and acceleration for each dynamic object in order to model its driving behavior. Dynamic objects are position in the virtual world of the environment models with specific velocity and acceleration. Over time, parameters of dynamic objects change inherently, like the

position due to the velocity, or parameters change at specific time stamps or even at specific positions by so-called *triggers* in VTD.
Relations between dynamic objects as well as between dynamic and static objects in the test scenarios have to preserved by the environment models of the simulations. The positioning of static and dynamic objects and the behavior of dynamic objects have to be aligned over time. For example, an additional lane, like an on-ramp, has to be assumed adjacent to the rightmost lane and the corresponding lane marking between these lanes has to be dashed if a vehicle is supposed to change onto the rightmost lane in the test scenario. Furthermore, the behavior of dynamic objects in the simulations must maintain all relations and restrictions on the behavior of dynamic objects in the test scenarios. For example, the estimated velocities for dynamic objects in the environment models still have to preserve the relative velocities by the corresponding test scenarios.

7.2.3.3.1. Intermediate Data Model

Test scenarios abstract completely from parameters or define different domains for identical semantic parameters which are required by the simulation frameworks for the environment modeling. Parameters describing the scenery, its roads, static objects, and dynamic objects in the environment models may not sufficiently be represented in the required detail by the test scenarios. For example, the runtime monitoring of the lane change assistant has not recorded the curvature of the street because the runtime monitoring perceives the road as straight segment without any curvature. For dynamic objects, the runtime monitoring only recorded their relative velocities to the automated ego vehicle and not their absolute velocities which are required for the environment modeling by the simulation framework VTD.
Nevertheless, the environment models of simulation frameworks have to sufficiently model the sceneries and dynamic objects of the corresponding test scenarios in order to enable to verify and validate the autonomous vehicle systems in the system verification for simulations of these test scenarios. Missing but required parameters for the modeling for scenery, static objects, and dynamic objects in environment models have to be calculated from available abstract parameters of the test scenarios.
Insufficient or missing parameters for the definition of roads in the environment models can be resolved by modeling different road topologies and attach the behavior. These road topologies have different values for the required parameters which are not represented in the test scenarios (cf. [Oli+16; Zof+15]). National and international standards can be considered to limit the range of values for these parameters in order to model realistic and relevant road topologies. For example, the curvature of German highways is limited by 280m [For08; Ric16]. The real roads in the corresponding test drive for the test scenarios can only be modeled if all required parameters, e.g., the curvature, are captured in the test scenarios.
Parameters for the behavior of dynamic objects in the environment models cannot be resolved by modeling predefined all possible behavior for all combinations of dynamic objects in the abstract representation of the test scenarios in advance. The combinatoric

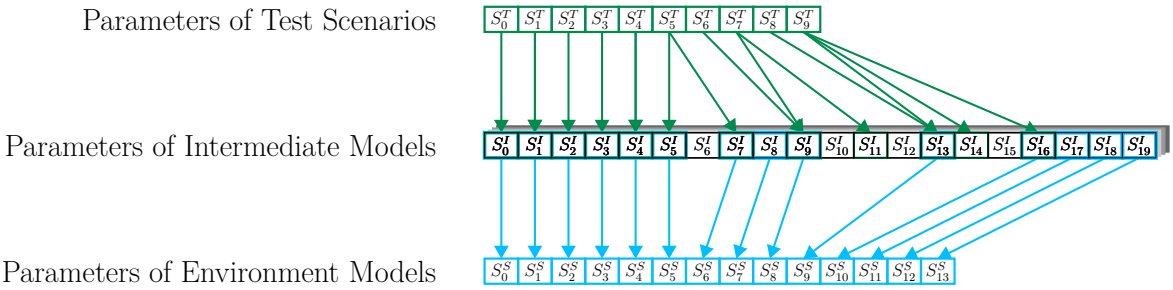


Figure 7.14.: Mapping of parameters between test scenarios, intermediate models, and environment models.

possibilities are too vast — especially if the number of possible dynamic objects in the abstract representation of the test scenarios is not limited.

The parameters of the environment models for the behavior of dynamic objects have to be calculated from available parameters of the test scenarios or estimated based on additional heuristics, e.g., risk-based heuristics (cf. [MG14]). In the case study on the lane change assistant, VTD requires absolute velocities for the modeling of vehicles. The absolute velocities of vehicles are calculated for the simulations from the recorded relative velocity of the automated ego vehicle and the relative velocity of each vehicle.

For the calculation of values for missing parameters and parameters with mismatching value domains in the environment model, the parameters from the test scenarios have to be mapped to the required parameters of the environment models for each simulation framework. The calculations may include simple inversions of the calculation from *input abstraction* of the runtime monitoring framework (cf. Section 6.1.1) or complex estimations based on physical models, e.g., vehicle dynamics.

An approach to address the mapping of parameters between test scenarios and environment models is the introduction of an intermediate model (cf. Fig. 7.14). This intermediate model aggregates all predictable parameters which environment models of any simulation framework might require. Therefore, the set of parameters in the intermediate model contains more parameters than any individual simulation frameworks requires for the environment modeling. The intermediate model may contain multiple parameters for one real-world property but each parameter with different value domain. One example of such parameters would be the different definition of object position and object orientations in diverging coordinate systems.

The *intermediate model* decouples the refinement of test scenarios from their modeling in specific simulation frameworks (cf. Fig. 7.14). Different simulation frameworks can be used for the simulation of test scenarios without adapting the calculations in the refinement of test scenarios. Fixed mappings are defined between the parameters of the *intermediate model* and the environment model parameters of each simulation framework. Parameters which required by the environment models of each simulation framework are individually mapped to the *intermediate model*. Each parameter of the simulation framework is represented by exactly one parameter in the *intermediate model* but not

each parameter of the *intermediate model* has to be mapped to a parameter of the simulation framework.
The transformation from test scenarios to the *intermediate model* can be defined independently from their modeling as environment models in simulation frameworks. As shown in Fig. 7.14, the relation between parameters of test scenarios and the intermediate model is *n:n*. One parameter of the test scenarios can be used in calculations for multiple parameters of the intermediate models and parameters of the *intermediate model* may require the consideration of multiple parameters from the test scenarios. Nevertheless, The calculation in the transformation of parameters from test scenarios to the *intermediate model* remain fixed as long as the intermediate model is not changed.
The possible expressiveness of simulations can be estimated before the transformation from test scenarios based on the parameters of the *intermediate model*. Parameters of the intermediate model which can be processed from the test scenarios can be compared to the parameters of the intermediate model which are required for simulations by the simulation framework. Different types of simulations may require different information to be recorded by the test scenarios.
For example, weather can only be simulated if the runtime monitoring and test scenarios have recorded any information about the current weather. Another example would be the recording by surrounding traffic on neighbor lanes. While the recording of preceding objects in the current lane would be sufficient for an ACC, this information would not be sufficient for the testing of a lane change assistant in simulations.
The transformation of parameters from the test scenarios into parameters of the intermediate model may result in *fuzzy parameters* for the intermediate model. The next section describes the resolution of *fuzzy parameters* for the intermediate model in order to use these parameters in the environment models of the different simulations frameworks.

7.2.3.3.2. Fuzzy Model Parameters

The transformation of parameters with abstract domains from the test scenarios into parameters of the intermediate model may result in *fuzzy parameters* for the intermediate model which cannot directly be incorporate in the environment models of the simulation frameworks. *Fuzzy parameters* contain intervals of values but not single absolute values. The *input abstraction* of the runtime monitoring framework processes multiple physical values for data parameters of the autonomous vehicle systems to identical abstract values of corresponding parameters in the *abstract situation*. The inherent abstraction of the *abstract situation* is responsible for the intervals for *fuzzy parameters* in the intermediate model in the *test scenarios*. In the case study on the lane change assistant, the position of dynamic objects is defined by their position on lanes and relative to the automated ego vehicle and not as absolute coordinates in the global system of VTD.

Definition 7.7 (Fuzzy Parameter)**.** *A fuzzy Parameter is a parameter which does not contain a single absolute value but an interval of possible values. In the case of the runtime monitoring, this interval resides from the inherent abstraction of the runtime monitoring framework.*

Simulation frameworks require for each parameter of their environment models exactly one single absolute parameter value. For any *fuzzy parameter* of the intermediate model which are required by the simulation framework, reasonable candidates have to be chosen from the interval of abstract values. These candidates can be determined from the abstract value intervals arbitrarily or structured by, e.g., equivalence partitioning under consideration of value limits or heuristics — e.g., risk-based heuristics (cf. [Agg14; MG14; MPB09; Sch+14]). As a result, a set of concrete values is determined as candidates for each fuzzy parameter.

The resolution of *fuzzy parameters* substitutes the intermediate model with the *fuzzy parameters* by a new set of intermediate models which solely contain concrete values for all parameters (cf. Fig. 7.14). Each *intermediate model* of this set represents a combination of a single candidate for each *fuzzy parameter*. The non-fuzzy parameters are identically used in all *intermediate models*. These *intermediate models* describe the same abstract test scenario but with different values for parameters which have not been recorded by the runtime monitoring framework. These parameterizations of a test scenario are called *test cases*. A test case can directly be transformed into an environment model for simulation in the system verification. Multiple test case can be defined for one test scenario.

For the lane change assistant, test cases were manually defined for each of the seven test scenarios. A basic set of road layouts has been defined which could be used to model the changes in the scenery in the test scenarios. These layouts consist of multi-lane highways with different patterns of lanes and different markings types. The main road layout for the modeling of new test cases from the recordings has been a highway with three lanes and standard markings (solid at the sides and dashed between the lanes).

Fuzzy parameters in the *abstract situations* from the runtime monitoring of the lane change assistant have been resolved for the modeling of the test scenarios in VTD by manually selecting one parametrization for each test scenarios from all possible combinations of candidate values for *fuzzy parameters* in the test scenarios. For example, the runtime monitoring framework has only recorded velocities of dynamic objects relative to the automated ego vehicle, but VTD requires absolute velocities for each dynamic object. An intermediate model has not been defined in the case study (cf. Chapter 8). The selected test cases have been manually modeled as environment models for simulations in VTD. Results from the verification of the lane change assistant in these simulations are presented in Chapter 8.

The following section assesses the evolution of autonomous vehicle systems and the generation of test scenarios and test cases based on results by the runtime monitoring during operation in the real world.

7.3. Assessment of System Evolution and Test Generation

The runtime monitoring of autonomous vehicle systems during operation in the real world contributes to the safety of these systems. The initiation of safety measure for unsafe system behavior and unknown traffic situations ensures the safety of autonomous vehicle systems during operation. The recordings of unsafe system behavior and unknown traffic situations enable the persistent improvement of autonomous vehicle systems. System faults and risky behavior are resolved in further development iterations. System limits are elaborated in extensive simulations. Despite these benefits also limitations have to be considered for conclusions on the safety of autonomous vehicle systems from results of the runtime monitoring during operation.

Prerequisites for accurate runtime monitoring results are the correct abstractions of system inputs and outputs by the runtime monitoring framework (cf. Section 6.4). Results about the correctness from operation in the real world and verification in simulations of autonomous vehicle systems can be compared to validate the correctness of the *input abstraction*. All *abstract situation* of the *tested situation knoweldge* must be verified in the simulations of the system verification and therefore have exhibit correct system behavior. Any encountered *incorrect* but *tested abstract situation* indicates a faulty implementation of the *input abstraction* or *abstract function*. The situation has been differently verified in the simulations by the same runtime monitoring framework as during operation in the real world. Therefore, a 0 percentage of *incorrect* and *tested* indicates the correctness of the implemented abstraction of traffic situations.

The complexity of test scenarios and test cases for the verification of autonomous vehicle systems is subject to the complexity in the definition of *abstract situations* from the specifications of these systems. Test scenarios and test cases can only model real-world objects which have been considered by the runtime monitoring and specifications of the autonomous vehicle systems. Undefined real-world objects, which are not reflected in the recordings of *abstract situations* by the runtime monitoring, are hidden from the test scenarios. The completeness of the system specification is essential for the validity of runtime monitoring results and test scenarios for the real world. The specification of autonomous vehicle systems has to be extended by encountered unknown objects in order to consider these objects in *abstract situations* and test scenarios.

Conclusion about the safety of autonomous vehicle system from these runtime monitoring results for the complete the real world have to consider the distribution of traffic situations in the corresponding test drives. The engineering approach can only define test scenarios for situations which have been monitored and recorded during operation in the real world. The occurrence of traffic situations in the real world follows a normal distribution. The runtime monitoring is likely to encounter the same traffic situations frequently during the operation in the real world. Everyday traffic situations are frequently encountered and recorded by the runtime monitoring during operation in the real world. Critical situations are less common in the real world and, therefore, less frequently recorded. The distribution of traffic situations during operation in the real world reflects itself in

the generated test scenarios for the verification of autonomous vehicle systems in the real world. Though critical situations provide the most impact for the verification and validation of autonomous vehicle systems the autonomous vehicle systems are more likely to be quickly improved and verified for everyday traffic situations than critical situations.

Statistically, the total number of critical situations increases with the number of driven test miles. Considerable test mileage is required to encounter and record sufficient numbers of critical situations. Large fleets of prototype vehicles may help to accumulate large mileages in short duration of time but would diminish the costs benefits of simulations in comparison to field operational tests for the verification of autonomous vehicle systems. The operation in the real world has to be systematically planned to increase the probability of encounter with less frequent but safety-critical traffic situations. The runtime monitoring can support this systematic planning by classifying the impact of individual roads for the safety verification of autonomous vehicle systems. Test drives can be planned to include a broad distribution of traffic situations.

The identification of test scenarios from traces of recorded *abstract situations* and the fuzzing of test scenarios by test cases can address the limitations of test scenarios due to the distribution of traffic situation. The test suite should comprehensibly verify the behavior of autonomous vehicle systems in a vast number of everyday and critical traffic situations with a small number of test cases in order to have a significant impact on the scope and validity of the system verification. Otherwise, the costs and time for the verification of autonomous vehicle would diminish the benefits of simulations in comparison to field operational tests.

The *situation graph* can help to create additional tests scenarios by combining previously unrelated sequences of *abstract situations*. Segments from different traces of *abstract situations* can be combined to form new traces and test scenarios. However, these new test scenarios may model unrealistic behavior by dynamic objects and unrealistic changes in the scenery because they have not been monitored and recorded in the real world. Checks for the realism of test scenarios have to be introduced in order to ensure the realism, validity, and efficiency of the system verification.

An impact on an efficient verification of autonomous vehicle systems has the fuzzing. Fuzzing of abstract parameters in test scenarios by combinations of concrete parameters in the test case enables to systematically and effectively explore the concrete parameter space for the system limits. Concrete parameter combinations have to be selected intelligently as candidates for the definition of test cases in order to sufficiently cover the parameter space of autonomous vehicle systems and their environments with a limited number of test cases.

Without proper realism checks and intelligent fuzzing, a large number of traffic cases, vast numbers of test scenarios and test cases would be created for the verification of autonomous vehicle systems. Would verify The behavior of autonomous vehicle systems would be verified for similar everyday traffic situations for long durations of time with little to none benefit for the safety assurance of these systems. Novel metrics have to be introduced to quantify and rate the impact of individual test scenarios and test cases for an impactful but efficient test suite.

Engineers manually implement the suggested improvements for autonomous vehicles systems based on the result of the runtime monitoring and system verification. Any system improvements can introduce new faults to the specification, design, and implementation of the autonomous vehicle systems. A systematic improvement process could support the correct implementation of system improvement and reduce the possibility of new system faults. The verification of the autonomous vehicle systems for existing and new test cases from the runtime monitoring results is inevitable. The test cases have to comprehensively verify the improvements of system faults and unsafe system behavior for autonomous vehicle systems under preservation of already correct and safe functionalities.
The following section presents the case study for the presented engineering approach on an industrial lane change assistant.

8. Case Study

The engineering approach (cf. Section 4.3) has been evaluated on a lane change assistant (cf. Section 3.1) in a project with academic and industrial partners. The setup and results from this evaluation are presented in the following sections.

8.1. Evaluation Setup

For the evaluation of the engineering approach (cf. Section 4.3), one of the project partners provided us with an industrial prototype of highway pilot. This highway pilot includes a lane change assistant (cf. Section 3.1) which has been used for this evaluation. In the course of this evaluation, two configurations are used. The first configuration utilizes the simulation framework VTD in order to simulate a virtual world in which the highway pilot operates (cf. Chapter 5). The second configuration substitutes the simulation framework with recordings of real-world tests drives. The simulation framework executed the lane change assistant in an *closed-loop* while the recordings only allowed an evaluation of the lane change assistant in an *open-loop* (cf. Section 2.1.5). The following sections describe each configuration in more detail.

8.1.1. Highway Pilot with Lane Change Assistant

Our project partner provided us with the software components and the software execution platform of a highway pilot as well as recordings from test drives with prototype vehicles on public roads. The highway pilot has included a lane change assistant alongside other vehicle functions, e.g., ACC or LKAS. Recordings of real test drives were used in the case study because prototype vehicles were not available for the evaluation of the operational part of the engineering approach (cf. Section 4.3).
The provided software of the highway pilot consists of all necessary functional components for the environment perception and decision making (cf. Fig. 3.3). Our project partners had implemented each component as ADTF filters. ADTF filters are essentially C/C++ dynamic libraries. We were unable to introduce any adjustments or changes to these software components of the lane change assistant because the project partners only had provided us with the binaries files of the ADTF filters but not with the corresponding sources.
The provided software of the highway pilot was envisaged for its usage in the simulation framework VTD and missed sensor, actuators, and components for the processing of the sensor and actuator data. The perfect object data from the simulations substitute the sensors and their preprocessing of sensor data while data for the vehicle actuators are

directly forwarded to the vehicle model of the simulations (cf. Fig. 5.3). Nevertheless, all provided software components of the highway pilot can be assigned to one of the three segments of the *system layer* — *preprocessing*, *function*, and *postprocessing* (cf. Fig. 5.1).

8.1.2. Runtime Monitoring Framework

The runtime monitoring framework (cf. Fig. 5.1) for the monitoring of the lane change assistant in simulations and during operation implements the requirements of the lane change assistant (cf. Section 3.2). Each requirement was transfered into a typed first-order logic formula (cf. Section 6.2.2.2.6). Based on these formulas, the *input abstraction*, *output abstraction*, *abstract function*, *situation monitoring*, *conformity oracle*, and *situation oracle* have been manually implemented as individual ADTF filters (cf. Section 6.2). The data structure of the *abstract situation* for the lane change assistant is presented in Fig. A.2. Figure A.1 shows the data structure of the *abstract target* in this case study.
The scope of the runtime monitoring in the evaluation was on the*lane change assistant*. Other functions of the highway pilot, e.g., ACC or LKAS, were not monitored in the evaluation. Two interfaces between runtime monitoring framework and highway pilot were defined for the access of the pilot's runtime data:

Input of the lane change assistant: The *input abstraction* receives the *situation* (cf. Fig. 3.9) at the interface between the components *function specific scene augmentation* and *situation assessment & situation predication* of the lane change assistant (cf. Fig. 3.3).

Output of the lane change assistant: The *output abstraction* is attached to the interface between the components *lane change assistant* and *aggregation* (cf. Fig. 3.3 in order to receive the processed *target point* by the lane change assistant (cf. Fig. 3.13).

In each processing cycle, runtime data via these two interfaces is forwarded to the *input abstraction* resp. *output abstraction* for the *qualitative* and *quantitative runtime monitoring* (cf. Section 6.2.3.4 and Section 6.2.3.5). The runtime monitoring framework evaluates the lane change assistant based on the provided runtime data and recorded faulty behavior of the lane change assistant and unknown *abstract situation* in log files (cf. Section 6.2.3.6).

8.1.2.1. Simulation Framework

The simulation framework VTD was employed in the evaluation for system verification of the lane change assistant (cf. Section 8.2.1) and for the imitation of the system's operation in the real world on test tracks in random traffic (cf. Section 8.2.2) The technical configuration of the simulation consisted of two separated computers (cf. Fig. 8.1). Computer 1 executed the highway pilot in ADTF and the simulation framework VTD

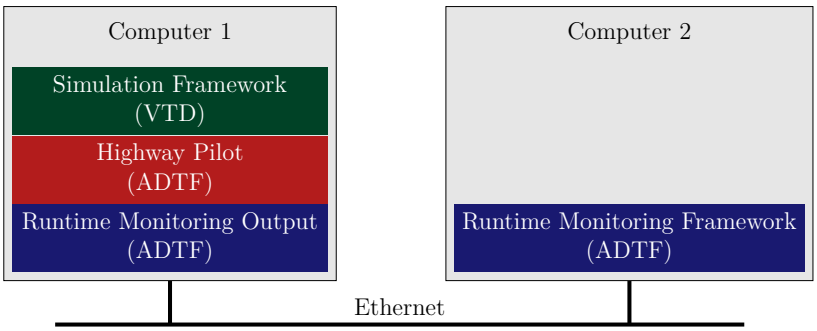


Figure 8.1.: Physical evaluation setup.

while Computer 2 executed the runtime monitoring framework in ADTF. Both computers were connected via a Ethernet network.
For the runtime monitoring of the lane change assistant, two components of the runtime monitoring framework (cf. runtime monitoring output in Fig. 8.1) were integrated into the ADTF instance of the highway pilot on Computer 1. In each processing cycle, these components collect the runtime data at the two interfaces and transfer the collected data via the Ethernet connection to the runtime monitoring framework on Computer 2. The runtime monitoring framework on Computer 2 evaluates the behavior of the lane change assistant based on the provided data and records faulty behavior and unknown *abstract situation*.
The *closed loop* between the simulation in VTD and the highway pilot was established on Computer 1 (cf. Section 2.1.5). In each processing cycle, the highway pilot receives the current environmental situation from the simulation and processes corresponding maneuvers to safely drive the automated ego vehicle in the simulation. Instead of fuzzy sensor data, exact positions of static and dynamic objects in the simulation are used as inputs for *scene modeling* of the highway pilot. The final maneuver as the output of component *aggregation* is transferred via the model of the vehicle dynamics back into the simulation. The vehicle dynamics model substitutes the real actuators of the vehicle in the simulations.

8.1.2.2. Recordings from Real World Test Drives

Recordings from test drives with prototypes vehicles on German highways A 2 and A 39 were used in addition to the simulation framework *VTD* because no prototype vehicle was available for the deployment of the runtime monitoring framework and its operation in the real world. The recordings had been recorded prior by one of the project partners before the case study. Even though the runtime monitoring framework could not be evaluated inside a real vehicle while operating in the real world, these recordings have enabled a more realistic evaluation of the runtime monitoring during operation than the simulations on virtual test tracks with random traffic.
The prototype vehicle which was used for the recording had two standard computers in the trunk for the processing of the highway pilot (cf. Fig. 3.16). The highway pilot in

the prototype vehicle was implemented as ADTF filters in the same version as used in the VTD simulations. The usage of ADTF enabled to replay of any recorded signal data of the highway pilot in the case study.
In the experiments with the recordings, the highway pilot was evaluated in an *open-loop* (cf. Definition 2.21) because the simulation framework VTD was substituted by the recordings. Recorded sensor data from the real sensors were used as input to the *scene modeling* of the highway pilot (cf. Fig. 3.3).
The output of the highway pilot — the final *target point* by the component *aggregation* (cf. Fig. 3.3) — remained unused for the recordings. The recordings did not offer the possibility to change a virtual environment in reaction to the behavior of the vehicle resp. the highway pilot. However, the behavior of the highway pilot in this experiment was consistent in its behavior in the prototype vehicle because the versions in the prototype vehicle and the simulations were identical.

8.2. Evaluation Results

In the course of this thesis, three conjectures have been revealed for the engineering approach (cf. Section 4.3). These conjectures have been addressed by the result of the case study on the lane change assistant. The three conjectures are:

C1. Inadequacies by the inherent abstraction of the runtime monitoring framework will reveal themselves in results of the runtime monitoring.

C2. Manually modeled traffic scenarios, and randomized simulations lack necessary realism, not for the validation of autonomous vehicle systems.

C3. Tests based on runtime monitoring results in realistic traffic allow achieving a satisfiable coverage of real-world situations.

Two experiments were executed on the lane change assistant for the investigation of these three conjectures. These two experiments represent the workflow of the engineering approach (cf. Fig. 4.1) and distinguish themselves into runtime monitoring within the system verification of the lane change assistant and runtime monitoring of the lane change assistant during operation. For the evaluation, the runtime monitoring of the lane change assistant during operation, simulations of randomized traffic in VTD and recordings from real-world test drives on Germany Highways were introduced.
The experiments in this evaluation are:

Experiment E1. Verification of lane change assistant by the runtime monitoring framework and recording of the *tested situation knowledge* in simulations of 22 manually modeled test case as well as runtime monitoring of the lane change assistant with the *tested situation knowledge* in simulations with randomized traffic and recordings from real-world test drives.

Experiment E2. Training of the runtime monitoring framework in simulations of seven "real worlds" test cases and runtime monitoring of the lane change assistant in the recordings with the extensive *tested situation knowledge* by the seven additional test cases.

The results from these experiments are presented in the following sections.

8.2.1. Experiment E1

The first experiment *E1* used manually modeled test cases to verify the lane change assistant for simulations in VTD and to record encountered *abstract situations* as *tested situation knowledge* for the runtime monitoring of the lane change assistant during operation. As no prototype vehicle had been available for the deployment of the runtime monitoring framework and its application in the real world, simulations in VTD on virtual tracks with randomized traffic and recordings from test drives on German highways were used. Experiment *E1* has addressed the conjunctures *C1* and *C2*.

8.2.1.1. System Verification with Manually modeled Test Cases

Experiment E1 commenced with the training of the runtime monitoring framework for a set of manually modeled test cases. One of the project partners manually modeled eight test scenarios based on the requirements of the lane change assistant from Section 3.2. Each test scenario defined the basic road layout, the number of dynamic objects, and their basic behavior. An overview of the eight test scenarios for the lane change assistant is given in Appendix A.2.
For the system verification of the lane change assistant, the eight test scenarios were refined and parameterized by 24 test cases (cf. Section 7.2.3.3). Each test case finalizes the road layout and behavior of dynamic objects by defining mutable physical parameters, e.g., the curvature of the road or the velocity of dynamic objects (cf. Appendix A.2). In addition to the definition of road layout and object behaviors, the expected behavior for the lane change assistant is defined in each test case.
The lane change assistant was verified for each test case by a simulation in VTD. A *environment model* defined for each test case the virtual world in the corresponding simulation (cf. Section 6.3.1). The duration of the test cases resided under 60 seconds — with one exception — accumulating to a total test duration of 14 minutes and 48 seconds. The behavior of the lane change assistant in these simulations has been evaluated in two ways — manually and by the runtime monitoring framework. The evaluation results are presented in the following sections.

8.2.1.1.1. Results of Manual Evaluation

The exhibited behavior by the lane change assistant in the simulations of each test case has been manually evaluated by one of the project partners in comparison to the expected behavior of each test case. As shown in Table 8.1, 22 test cases from the 24 defined test cases have been evaluated as successful.

Table 8.1.: Result of the manual evaluation for the manually modeled test cases.

Test Case	Test Scenario	Duration	Manual Verdict
Test Case 1	Scenario 1	50.115 sec	Success
Test Case 2	Scenario 1	61.175 sec	Success
Test Case 3	Scenario 1	50.715 sec	Success
Test Case 4	Scenario 2	40.135 sec	Success
Test Case 5	Scenario 2	40.175 sec	Success
Test Case 6	Scenario 2	40.115 sec	Success
Test Case 7	Scenario 3	40.175 sec	Success
Test Case 8	Scenario 3	40.115 sec	Success
Test Case 9	Scenario 3	40.115 sec	Success
Test Case 10	Scenario 3	40.115 sec	Success
Test Case 11	Scenario 4	40.115 sec	Success
Test Case 12	Scenario 4	40.115 sec	Success
Test Case 13	Scenario 4	40.115 sec	Failure
Test Case 14	Scenario 5	21.335 sec	Success
Test Case 15	Scenario 5	25.735 sec	Success
Test Case 16	Scenario 5	27.995 sec	Success
Test Case 17	Scenario 5	28.475 sec	Success
Test Case 18	Scenario 6	24.315 sec	Success
Test Case 19	Scenario 6	28.335 sec	Success
Test Case 20	Scenario 7	27.475 sec	Success
Test Case 21	Scenario 7	38.715 sec	Success
Test Case 22	Scenario 7	28.835 sec	Success
Test Case 23	Scenario 8	40.175 sec	Failure
Test Case 24	Scenario 8	33.755 sec	Success
Total Test Duration		14 min 48 sec	

The lane change assistant failed to exhibit the anticipated behavior in 2 of the 24 test cases. For the third test case of test scenario 4 (cf Table A.4), the highway pilot was not able to establish the initial position of the test case by positioning and driving the automated ego vehicle with a predefined offset to its lane center. Changes to the source code of the highway pilot would have been necessary. The first test case of scenario 8 (cf. Table A.8) has been evaluated as unsuccessful by project partners, because the lane change assistant failed to change its lane onto an off-ramp from the mid lane of a three-lane highway through a convoy of vehicles driving on the right highway lane with short gaps between the vehicles. This fault was communicated to the developers of the lane change assistant and resolved in a later version of the system.

8.2.1.1.2. Results of the Runtime Monitoring

In addition to the manual evaluation by project partner, the runtime monitoring framework has evaluated the correctness and safety of the lane change assistant (cf. Section 5.4) in the simulations for the 24 test cases. The foundation for this evaluation were the requirements for the lane change assistant from Section 3.2.
In each processing cycle of the lane change assistant, the runtime monitoring framework processed the current *abstract situation* and evaluated all conditions in the system requirements for this *abstract situation*. A test case has been evaluated as a failure by the runtime monitoring framework if a condition in any *abstract situation* was violated.
The results of the runtime monitoring in the simulations of the manually modeled test cases are presented in Table 8.2. In addition to the verdict of the runtime monitoring framework, Table 8.2 also shows the numbers of encountered, unique, recorded, and faulty *abstract situations* for each test case. The numbers of recorded *abstract situations* represent the amount of *abstract situation* which have been recorded by the runtime monitoring for the *tested situation knowledge* in each test case (cf. Section 8.2.1.1.3). The test cases have encountered in average approx. $1,678$ total situations per minute, approx. 20.78 unique situations per minute, and recorded approx. 18.58 situation per minute for the *tested situation knowledge*. The test cases have verified the lane change assistant in simulations for a total duration of 14 minutes and 48 seconds and have accumulated 25,155 *abstract situations* in total and 301 total unique *abstract situations* — including duplicated *abstract situations* (cf. Table 8.2).
The runtime monitoring framework has evaluated the behavior of the lane change assistant in the simulations with regard to its requirements (cf. Section 3.2) for 8 test cases as unsafe and 16 test cases as safe and correct. Three main reasons can be identified for the eight failed test cases. Real faults in the execution of lane changes represent the first reason. The behavior of lane change assistant in test cases 7, 8, and 10 have violated in more than 100 situations at least one requirement of the lane change assistant (cf. Section 3.2).
The second reason for failed test cases was the missing information about positions of on-ramps and off-ramps. The lane change assistant did not provide the runtime monitoring framework with sufficient data about the position and length of on-ramp and off-ramps. This missing information prohibited any evaluation of test cases with the restriction on lane changes in proximity of these ramps. The runtime monitoring framework evaluated the behavior of the lane change assistant in test cases 15, 16, and 17 for 64 to 85 *abstract situations* as unsafe. This inadequacy also has affected successful test cases. Even though on-ramps were not correctly recognized for test cases 20, 21, and 22, the behavior by the lane change assistant in these test cases complied with its system requirements (cf. Table 8.2).
The third reason for failed test cases is a diverging interpretation of lane markings. The lane change assistant has interpreted *unknown markings* as non-existing and has allowed lane change between lanes over these *unknown markings*. The runtime monitoring framework evaluated *unknown* markings as *restricted* markings and denied any lane changes between lanes with *unknown* markings between them. As result, all lane changes

Table 8.2.: Result by the runtime monitoring for the initial set of test cases.

Test Case	Result	Situations				Remarks
		Total	Unique	Faulty	Recorded	
Test Case 1	Success	1233	15	0	14	
Test Case 2	Success	1411	6	0	4	
Test Case 3	Success	1438	19	0	17	
Test Case 4	Success	1201	14	0	13	
Test Case 5	Success	1152	12	0	11	
Test Case 6	Success	1360	4	0	4	
Test Case 7	Failure	1360	16	103	14	failure while lane changing
Test Case 8	Failure	1341	15	107	13	failure while lane changing
Test Case 9	Success	1315	16	0	11	
Test Case 10	Failure	1111	11	110	10	failure while lane changing
Test Case 11	Success	1152	14	0	12	
Test Case 12	Success	1353	20	0	16	
Test Case 13	Success	1155	14	0	12	
Test Case 14	Success	476	5	0	4	
Test Case 15	Failure	593	8	82	8	off-ramp not recognized
Test Case 16	Failure	734	14	84	13	off-ramp not recognized
Test Case 17	Failure	728	16	57	16	off-ramp not recognized
Test Case 18	Failure	584	9	95	9	*unknown* marking interpreted as *restricted*
Test Case 19	Failure	763	17	96	17	*unknown* marking interpreted as *restricted*
Test Case 20	Success	638	5	0	5	on-ramp not recognized
Test Case 21	Success	1366	14	0	7	on-ramp not recognized
Test Case 22	Success	663	10	0	10	on-ramp not recognized
Test Case 23	Success	1270	25	0	24	
Test Case 24	Success	771	2	0	2	
Total numbers		25,155	301	266	734	incl. duplicates

across *unknown* markings have been evaluated as unsafe by the runtime monitoring framework (cf. test cases 18 and 19 in Table 8.2). In test cases 18 and 19, 93 resp. 95 *abstract situations* have been evaluated as unsafe by the runtime monitoring framework. The evaluation of the lane change assistant has revealed the importance of precise requirements. Misinterpretation and imprecision of requirements have to lead to the most failures of test cases for the lane change assistant in addition to the real failures in the execution of lane changes. Two test cases have been evaluated as a failure due to the diverging interpretation of *unknown* lane markings.
The runtime monitoring framework initially evaluated all lane changes — including correct and safe ones — as unsafe due to a misinterpretation of FR_2_3. The requirement FR_2_3 had initially required that the lane change assistant would have to prevent lane changes if the vehicle deviates more than 0.4 m from the center of its current lane. However, any lane change by the vehicle between two lanes inherently deviates more than $0.4\,m$ from the center of the current driving lane. As a result, the runtime monitoring framework evaluated the requirement FR_2_3 to be violated each time a lane change was performed. For further evaluation, requirement FR_2_3 has been redefined to not apply to lane changes which are in execution.
In addition to the qualitative evaluation of the behavior by the lane change assistant in the 22 test cases, the *quantitative runtime monitoring* also recorded the encountered *abstract situations* for later usage as *tested situation knowledge* in the runtime monitoring during operation.

8.2.1.1.3. Recording of Tested Situation Knowledge

In addition to the *qualitative runtime monitoring* of the lane change assistant, the runtime monitoring framework recorded the encountered *abstract situations* in each test for the usage as *tested situation knowledge* during operation in simulations with random traffic and real world recordings (cf. Section 8.2.1.2 and Section 8.2.1.3). In the simulation of the test cases, the runtime monitoring framework operated in the *recording state* (cf. Section 6.3.3) and has recorded in average approx. 28.88 *abstract situations* per second (approx. 0.81 *unique abstract situation* per second). The exact numbers of encountered and recorded *abstract situations* for each test case are presented in Table 8.2.
As apparent from the total number of *abstract situation* and number of *unique abstract situations*, single *abstract situations* occurred multiple times in the simulation of each test case. Only the unique *abstract situations* are of interest for the *tested situation knowledge*. Duplicated *abstract situations* have no benefit for the runtime monitoring during operation and can increase the processing time of the runtime monitoring due to longer searches for *abstract situations* in the *tested situation knowledge*. Therefore, duplicated *abstract situations* were eliminated from the set of rerecorded *abstract situation*. The remaining unique *abstract situations* were serialized and stored in an individual XML file for each test case (cf. Section 6.3.3).
The results in Table 8.2 show that not all unique *abstract situations* were recorded by the runtime monitoring framework. This problem is related to the implementation of the runtime monitoring framework, timing and processing shortcomings of ADTF, and

Table 8.3.: Test cases with *abstract situations* in the *tested situation knowledge*.

Test Case 1	Test Case 2	Test Case 3	Test Case 4	Test Case 5
Test Case 6	Test Case 9	Test Case 11	Test Case 12	Test Case 13
Test Case 14	Test Case 20	Test Case 21	Test Case 22	Test Case 23
Test Case 24				

the utilization of the computational resources. In particular, the serializing of *abstract situation* to XML files represented a bottleneck in comparison to the rate the runtime monitoring framework received data from the lane change assistant.
As described in Section 7.1.2, the *quantitative runtime monitoring* during operation requires the *tested situation knowledge* to contain solely *abstract situations* from test cases which have been evaluated as successful. As any improvement of the lane change assistant was impossible due to missing source files, only *abstract situations*, which had been recorded in successful test cases, were used as *tested situation knowledge* for the runtime monitoring during operation. Table 8.3 lists the 16 test cases whose *abstract situations* were used for the *quantitative runtime monitoring* of the lane change assistant during operation in simulations of VTD and recordings from real world test drives (cf. Section 8.2.1.2 and Section 8.2.1.3).
The decision about the use of test cases and their *abstract situations* for the *tested situation knowledge* was solely made based on the assessment by the *qualitative runtime monitoring* in these test cases. The manual verdict was not considered for the selection of test cases. Therefore, the *abstract situations* of *Test Case 24* were included in the *tested situation knowledge* for the operation.
The *tested situation knowledge* for the *quantitative runtime monitoring* of the lane change system during operation in simulations and recordings of real-world test drives consisted of 77 *abstract situations*. This number does not match the sum of recorded *abstract situations* for all considered test cases. Even though duplicates are not recorded in the simulations of the initial test cases, the combination of recording *abstract situations* from different test cases for the *tested situation knowledge* can still include identical *abstract situations*. These *abstract situations* are identified and filtered in the import process of the XML files. As a result, only one instance of each recorded *abstract situation* is included in the *tested situation knowledge*.
The results from the *quantitative runtime monitoring* of the lane change assistant during operation are presented in the next sections.

8.2.1.2. Runtime Monitoring at Operation in Simulations with Random Traffic

For the runtime monitoring of the lane change assistant during operation, the simulation framework VTD was used to model virtual test tracks and to generate random traffic. Two cyclic highways have been modeled as test tracks. Test track I (cf. Fig. 8.2a) represents cyclic test track with two lanes but only in one direction. The test track II (cf. Fig. 8.2b) is provided with VTD and represents a city with junctions, lights, and a

(a) Simulation Test track 1. (b) Simulation Test track 2.

Figure 8.2.: Virtual test tracks for the runtime monitoring during operation.

complete highway ring. The evaluation on the second test track was solely performed on the highway ring. The highway of the first test track is one directional and has two lanes which are interrupted by one lane segments. The highway ring of the second test track has permanently two lanes in both driving directions.

Vehicles were randomly spawned 200 units in front and behind the automated ego vehicle for the evaluation of the lane change assistant on these two virtual test tracks. Up to 20 vehicles were present in the vicinity of the automated ego vehicle as random traffic. Every new vehicle was spawned with randomized vehicle parameters, e.g., velocity and acceleration, for every vehicle leaving the corridor of 200 units in front and behind the automated ego vehicle. The vehicle parameters were not changed in the course of the simulations. Vehicles leaving this corridor were removed from the simulations. The random traffic required the lane change assistant to overtake random slower vehicles by autonomously executing lane changes arbitrarily.

The lane change assistant was evaluated for each virtual test track two times for a duration of approx. 10 minutes and has accumulated in average 18,365 *abstract situations* of whom 549 *abstract situations* were unique (cf. Table 8.4). Each simulation itself was unique due to the randomization of traffic. Therefore, the results from different simulations on the same virtual test track are not comparable with each other.

While the lane change assistant autonomously drove in the randomized traffic, the runtime monitoring framework evaluated the behavior of the lane change assistant regarding its requirements (cf. Section 3.2) and assessed the encountered *abstract situation* in relation to the recorded *abstract situation* in the *tested situation knowledge*. The results from the runtime monitoring of the lane change assistant in the simulations on virtual test tracks are presented in Table 8.4. The following sections discuss the results in more detail.

8.2.1.2.1. Results from the Quantitative Runtime Monitoring in Simulations

For the *quantitative runtime monitoring* in the simulations on the two virtual test tracks, the runtime monitoring framework operated in the *comparison state*. The *situation monitor* would evaluate if encountered *abstract situations* had been verified based on the

Table 8.4.: Coverage of traffic situations after training the situation monitor.

Measurement	Duration	Situations [#]		Tested [%]		Not Tested [%]	
	[Min:Sec]	Sum	Unique	Correct	Incorrect	Correct	Incorrect
Track I (1)	10:37	18,419	567	14.92	0.0	85.07	0.01
Track I (2)	10:30	18,484	624	12.19	0.0	87.81	0.00
Track II (1)	10:48	18,679	488	15.58	0.0	82.25	2.17
Track II (2)	10:38	17,879	518	16.74	0.0	82.63	0.63
A2	5:53	7,078	1,078	0.68	0.0	96.62	2.70
A39	6:34	7,885	974	1.57	0.0	96.45	1.98

tested situation knowledge of 77 *abstract situations* from the simulation of the manually modeled test cases (cf. Section 8.2.1.1.3).
The results of the *quantitative runtime monitoring* of the lane change assistant during operation in simulations on the two virtual test tracks are shown Table 8.4. The percentages of *abstract situations* in these simulations are given by the total numbers in the columns *Tested* and *Not Tested*. From approx. $18,365$ encountered *abstract situations* (approx. 549 unique *abstract situations*) in the simulations on the virtual test tracks with randomized traffic within approx. 10 minutes only 12.19 % to 15.58 % of *abstract situations* have been previously verified (*Tested*) in the simulations of the initial set of test cases. The percentage of encountered *abstract situations* which are not contained in the *tested situation knowledge* resides above 83 %. In comparison, the initial set of test cases accumulated $25,155$ *abstract situations* (301 unique *abstract situations*) overall in a total test duration of 14 minutes and 48 seconds (cf. Table 8.1 and Table 8.2). With the initial set of test cases, at least 10 % of *abstract situations* in the simulations on the two virtual test tracks have been verified.

8.2.1.2.2. Results from the Qualitative Runtime Monitoring in Simulations

As in the simulations for the initial set of the manually modeled test case, the runtime monitoring framework evaluated the correct behavior of the lane change assistant in the simulations on the two virtual test tracks with random traffic regarding its requirements (cf. Section 7.1.1). Any *abstract situations* in which the runtime monitoring framework logged wrong and unsafe behavior of the lane change assistant.
Table 8.4 displays the percentage of encountered *abstract situations* with correct and safe system behavior (*Correct*) resp. critical and incorrect system behavior (*Incorrect*) in relation to the percentage of *tested* resp. *not tested abstract situations* by the *quantitative runtime monitoring* (cf. Section 8.2.1.2.1).
For the total number of *Correct* resp. *Incorrect abstract situations* in each simulation, the corresponding sub-columns in the *Tested* and *Not Tested* columns have to be accumulated. The percentage for *correct abstract situations* has resided above 97.83 %. Less than 2.17 % of *abstract situations* which had not yet been tested have been evaluated as incorrect and unsafe by the runtime monitoring framework for all simulations on the virtual test tracks with random traffic.

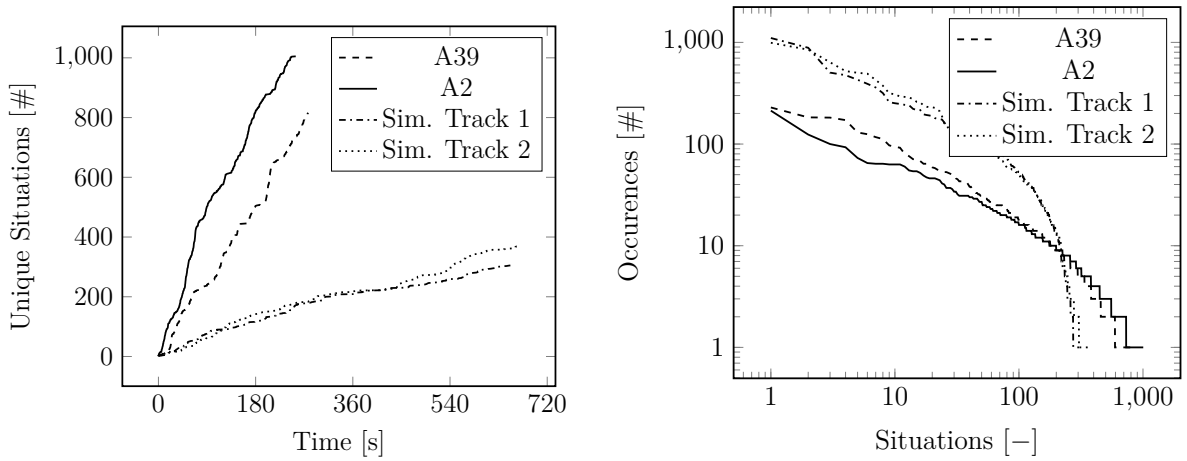
The percentage of *incorrect abstract situations* in the set of *tested abstract situations* is an important indicator for conjecture **C1.** — the soundness of the implemented abstraction in the runtime monitoring framework (cf. Section 5.3). There must not exist any *incorrect* but *tested abstract situations*.

> **Conjecture C1.** *Any inadequacies by the inherent abstraction of the runtime monitoring framework will reveal themselves in results of the runtime monitoring.*
> The number of *incorrect* but *tested abstract situations* is an indicator for inadequacies of the abstraction in the runtime monitoring framework. In any *tested abstract situations*, the behaviors of the autonomous vehicle systems have to match their previous correct and safe behavior in the simulations of the system verification. Inadequate implementations of the abstractions in the runtime monitoring framework allow for different system behavior to emerge in the same *abstract situations*. In the presence of any *incorrect* but *tested abstract situation*, the runtime monitoring framework and its inherent abstraction have to be revised.
> The 0 % of *incorrect* and *tested abstract situation* in the case study of the lane change assistant have indicated that an adequate abstraction had been implemented.

The significance of these simulations for the evaluation of the engineering approach is limited. Simulations on virtual test tracks with randomized traffic lack the necessary realism to sufficiently exhibit the extraordinary characteristic of the real word. For example, ideal sensor models, which were used in these simulations, do not sufficiently reflect the unique characteristic of real sensors and corresponding perception problems in the real world. Furthermore, simulations are unlikely to sufficiently considered all types of dynamic objects which autonomous vehicle systems encounter in the real world.
For the evaluation of the presented engineering approach, the unique characteristic of the real world and their impact on the autonomous vehicle systems are of particular interest. The engineering approach is envisaged to address the gap between realism and replicability between the virtual world and the real world. Recordings from real-world test drives on German highways were introduced in addition to the simulation on test tracks. The recordings addressed the extraordinary characteristic of the real world. The results from the runtime monitoring of the lane change assistant during operation in these recording are presented in the following section.

8.2.1.3. Runtime Monitoring at Operation in Recordings from the Real World

In addition to the simulations on virtual test tracks with randomized traffic, two recordings from test drives in prototypes vehicles on the German highways A2 and A39 were used for the evaluation of the lane change assistant during operation. The configuration for the usage of real-world recordings are documented in Section 8.1.2.2. All internal data signals of the highway pilot had been recorded in the test drives and were replayed in ADTF for the evaluation of the lane change assistant by the runtime monitoring framework in the case study.



(a) New situations over time.

(b) Distribution of frequencies.

Figure 8.3.: Distribution of frequencies and discovery of situations over time.

The two recordings from the test drives on the German highways A2 and A39 provide a more realistic environment for the operation of the lane change assistant than the simulations in VTD:

Highway A2 consists of three lanes in each driving direction and is subject to a high volume of road traffic.

Highway A3 consists of two lanes per direction hand is subject to mostly moderate road traffic.

The configurations of the highway pilot and the runtime monitoring framework in the two recordings were identical to the configurations in the simulations on virtual test tracks with randomized traffic. As in the simulations, 77 *abstract situations* from simulations of the manually modeled test cases were initially used as *tested situation knowledge* for the *quantitative runtime monitoring* of the lane change assistant in both recordings.
The recording on the highway A2 had a duration of 5 minutes and 53 seconds and has accumulated $7,078$ *abstract situations* in total ($1,078$ unique *abstract situations*). The recording from the A39 accumulated $7,885$ *abstract situations* in total (974 unique *abstract situations*) in 6 minutes and 34 seconds. The results for the runtime monitoring of the lane change assistant in both recordings are shown in Table 8.4. The following section discusses the results in more detail.

8.2.1.3.1. Results from the Quantitative Runtime Monitoring in Recordings

The runtime monitoring framework has recorded in the recordings *A2* and *A39* significant more unique *abstract situations* than in the previous simulations on virtual test racks with randomized traffic, even though the duration of the recordings was significantly shorter. As shown in Fig. 8.3a, new unique *abstract situations* are encountered significantly faster in both recordings than in the simulations. The simulations contained higher quantities

of single *abstract situations* than *abstract situations* in the recordings on the two highways (cf. Fig. 8.3b).
In addition to the increased discovery of *abstract situations*, the coverage of the 77 *abstract situations* by the *tested situation knowledge* has significantly dropped from at least 12.18 % in the simulations to 1.57 % for the recording *A39* resp. 0.68 % for the recording *A2* (cf. Table 8.4). The total numbers of encountered *abstract situations* in the two recordings (approx. 7481.5 *abstract situations* in average) resided below half of the numbers encountered in the simulation on virtual test tracks with random traffic (approx. 18,365 encountered *abstract situations*). The more substantial reduction for the recording on the highway A2 is subject to the number of lanes for this highway. All initial test cases model only two lanes. Recorded *abstract situations* in the simulations of these test cases are more likely to occur in the recording on the A39 than in the recording on the A2.

8.2.1.3.2. Results from the Qualitative Runtime Monitoring in Recordings

In addition to the reduced coverage of the *tested situation knowledge* in the recordings, the larger numbers and variations of *abstract stations* in both recordings lead to an increased percentage of incorrect *abstract situations* (cf. Table 8.4). The percentage of incorrect *abstract situations* has increased to 1.98 % for recording *A39* resp. 2.70 % for recording *A2*. These percentages have been a significant increase considering the lower number of total *abstract situations* in both recordings in comparison the numbers of *abstract situations* in the simulations on virtual test tracks with random traffic. In both recordings, there are no *abstract situations* which have been evaluated as *incorrect* but had been successfully *tested* in the simulations of the initial test cases.
The results of the runtime monitoring in the two recordings with the *tested situation knowledge* from the manually modeled test cases outline the conjecture **C2**.

> **Conjecture C2.** *Randomly simulated and manually modeled traffic scenarios lack necessary realism not for the validation of autonomous vehicle systems.*
> The results for the *qualitative* and *quantitative runtime monitoring* of the lane change assistant in the recordings of test drives on the German highways A2 and A39 show that manually modeled test cases are not able to sufficiently represent real-world traffic. The results of the runtime monitoring significantly diminish from simulations on test tracks to recordings from real-world test drives (cf. Table 8.4).

The difference in the complexity of the manually modeled test scenarios and the complexity in the real world is the reason for the shortcomings by the manually modeled test scenarios and test cases for the coverage of *abstract situations* in the recordings of real-world test drives on highways A2 and A39. Test engineers commonly verify the system for its requirements in scenario-based testing by defining test scenarios with the lowest necessary complexity (cf. Section 8.2.1.1.2). These scenarios are less likely to occur in the real world.

The coverage reduction for the *tested situation knowledge* and the increase of faulty *abstract situation* in the recordings on the German highways have encouraged the definition of additional test scenarios and test cases with realistic traffic from the log files of the runtime monitoring framework for these recordings. The next section presents the second experiment in which the lane change assistant was evaluated for seven additional test scenarios and test cases.

8.2.2. Experiment E2

The runtime monitoring of the lane change assistant during operation for the two recordings on the Germany highways A2 and A39 has disclosed that manually modeled test case based on system requirements do not sufficiently represent traffic situations in the real world. As a consequence of this result, seven additional test case were modeled based on the logs of the runtime monitoring framework in the recording on the highway A2 (cf. Section 8.2.2.1). The modeling of test cases from the results of the runtime monitoring during operation concludes the round-trip of the presented engineering approach (cf. Section 4.3.3).
Finally, the lane change assistant was verified in simulations of these seven test cases. The encountered *abstract situations* in these simulations were recorded by the runtime monitoring framework for the *tested situation knowledge* (cf. Section 8.2.2.2). The runtime monitoring framework supervised the operation of the lane change assistant in the recording on the highway A2 with the extended *tested situation knowledge* by the recorded *abstract situations* from the additional test cases (cf. Section 8.2.2.3).

8.2.2.1. Modeling of Test Cases from Runtime Monitoring Results in Recordings

Seven test cases were defined based on the observed but untested *abstract situations* in the recording from the highway *A2*. The recording from the A39 was not considered for the modeling of test cases. The modeling of these realistic test cases was manually performed similarly to the approach in Section 7.2.3. First, seven adjoint test scenarios were identified in the log data in the recording *A2*. Each test scenario was refined by one test case with one arbitrary parametrization of situation parameters (cf. Section 7.2.3.3).
The definition of the test scenarios commenced with the analysis of the log file from the runtime monitoring of the lane change assistant during operation in the recording on the highway A2. In the lists of *abstract situations* in this log file, changes of the scenery as well as the behavior of dynamic objects were analyzed. For the scenery, one fixed road layout — a highway with three lanes per direction — was defined for all seven test scenarios. The behavior of dynamic objects was defined by manually identifying all maneuvers of dynamic objects between two *abstract situations*. Multiple possible sequences of *abstract situations* and maneuvers by dynamic objects were identified from the log files (cf. Section 7.2.3.1). One of these sequences was arbitrarily selected for each test scenario. A *situation graph* was not created in this analysis (cf. Section 7.2.3.1), but ghosting objects were taken into account (cf. Section 7.2.3.1.3).

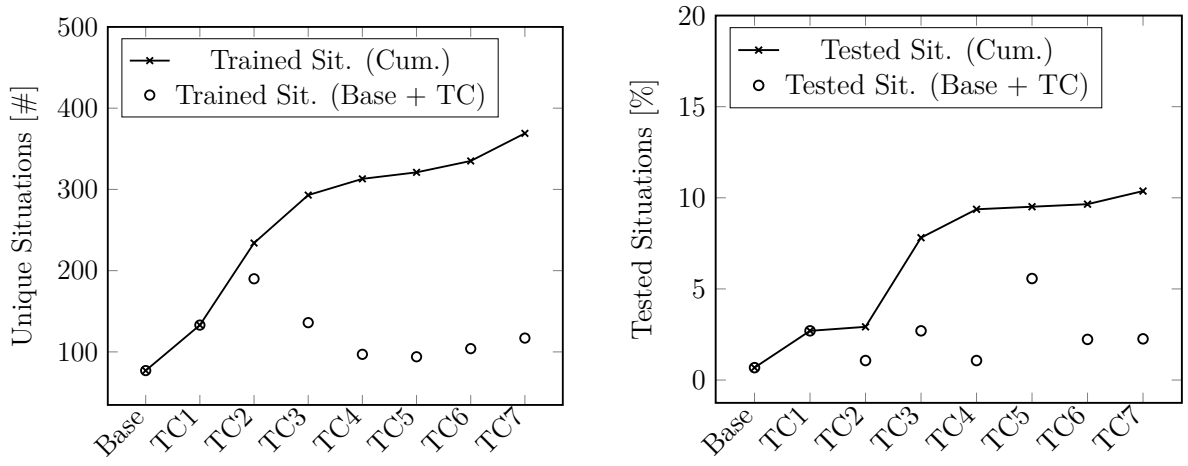
Table 8.5.: Runtime Monitoring Result from real world test cases.

Test Case	Situations			
	Total	Unique	Faulty	Recorded
Test Case 1	5675	69	0	57
Test Case 2	4582	125	0	113
Test Case 3	4771	65	0	59
Test Case 4	6926	43	0	27
Test Case 5	2757	31	105	27
Test Case 6	3878	37	110	33
Test Case 7	2550	53	217	40

The length of the test scenarios was manually decided based on the consistency of behaviors by dynamic objects throughout the *abstract situations*. The test scenarios were sliced at transitions between *abstract situations* where an *abnormal* behavior by a dynamic object was identified.
Appendix A.3 presents detailed information about the seven additional test scenarios. Each test scenarios contains in average ten dynamic objects in addition to the automated ego vehicle. These dynamic objects perform more than ten maneuvers in each test scenario. This practice ensures a consistent and realistic behavior of dynamic objects throughout all seven test scenarios.
The seven test scenarios do not define all necessary parameters for simulations in VTD. The missing parameters for modeling *environment models* in VTD were manually interfered and defined by one test case for each test scenario (cf. Section 7.2.3.3). The simulations of the seven test cases exhibited a significantly increased complexity in comparison to the initial set of manually modeled test cases.
The results of the runtime monitoring framework in the simulation of the seven test cases in VTD are presented in the next section.

8.2.2.2. Results from the System Verification with the Realistic Test Cases

The results from the runtime monitoring of the lane change assistant in the simulations of the seven test cases are presented in Table 8.5. The seven test cases encountered in average 4448 *abstract situations*. The numbers of *unique abstract situations* range from 31 up to 125. The reduction of recorded *abstract situations* is subject to the same timing problems and resource limitation by ADTF as for the initial set of manually modeled test cases (cf. Section 8.2.1.1.3).
For three of the seven test cases, the lane change assistant exhibited an incorrect behavior. The runtime monitoring recorded at least 110 *faulty abstract situations* in the simulation of test cases 5, 6, and 7. The missing source files of the lane change assistant made it impossible to resolve this faulty behavior. The small number of additional test cases required the usage of recorded situations from faulty test cases for the runtime monitoring during operation (cf. Fig. 8.4a).



(a) Simulated unique abstract situations.

(b) Coverage of observed situations.

Figure 8.4.: Coverage after training monitors with additional test cases generated from observed situations.

Figure 8.4a compares the cumulative increase of recorded *abstract situations* (line) with the increase of recorded *abstract situations* achieved by individual test cases (circles) for the re-training of the *tested situation knowledge* in simulations of test cases (*TC1 – TC7*).

The numbers of recorded *abstract situations* per test case (circles) contains the previous number of 77 *abstract situations* from the initial test cases (cf. Section 8.2.1.1.3) and the additional *abstract situations* from each new test case. While the runtime monitoring has recorded significant more *abstract situations* for test cases *TC 1* and *TC2* than the base number from the initial set of test case, test cases *TC4* and *TC5* have added only small numbers of new *abstract situations*.

The accumulated number of recorded *abstract situation* for the *tested situation knowledge* (line) significantly increases from initial 77 *abstract situations* to 369 situations with addition of the seven realistic test cases (cf. Fig. 8.4a). The cumulated number (line) for each test case (*TC1 – TC7*) contains the recorded *abstract situations* of all previous test cases — including the base number from the initial set of manually modeled test cases — under elimination of duplicated *abstract situations*. The encountered *abstract situations* in independent simulations of each test case correlate with the increase of the *tested situation knowledge* of all test cases accumulated (cf. Fig. 8.4a).

The seven additional test cases significantly increase the number of verified *abstract situations*. These *abstract situations* were are used as *tested situation knowledge* for the runtime monitoring of the lane change assistant during operation. The next section describes the runtime monitoring of the lane change assistant during operation in the same recording on the highway A2 with the increased *tested situation knowledge* of 369 *abstract situations*.

8.2.2.3. Runtime Monitoring at Operation with knowledge of Realistic Test Cases

For the impact of realistic test cases, the lane change assistant was reevaluated for the recordings $A2$ with the addition of the 369 recorded *abstract situations* from the seven additional test cases. Simulations on the virtual test track with random traffic were not considered for this re-evaluation due to their limited realism.
The small number of modeled realistic test cases required the consideration of *abstract situations* from failed test case for *tested situation knowledge* in this experiment (cf. Table 8.5). Therefore, the number of situations in the *tested situation knowledge* has increased from 77 to 369 *abstract situations* with the seven realistic test cases — including the faulty test cases 5, 6, and 7.
The results from the re-evaluation of the lane change assistant in the recording on the German highway A2 with the seven additional test cases is shown in Fig. 8.4b. The graph compares the cumulative percentage of *tested abstract situations* (line) with the changes of *tested abstract situations* by individual test cases (circles). The set of 369 *abstract situations* in the *tested situation knowledge* increased the coverage of *abstract situations* from 0.68% by the initial test cases to 10.37%.
The coverage of *abstract situations* in the recording $A2$ by individual test cases does not directly reflect in the cumulated coverage(cf. Fig. 8.4b). Test case 3 (TC3) has an average coverage of *abstract situations* in the recording but impacts the cumulate coverage significantly more than the other test cases. Test case 5 (TC5) has a very high percentage of *tested abstract situations* but the cumulated percentage does not significantly increase with the addition of test case 5. While any previous test cases, i.e., test cases 1 and 2, have not encountered the *abstract situations* from test case 3 (TC3), the predecessors of test case 5 (TC5) almost completely covered its *abstract situations.*
The number of *faulty abstract situations* for the re-valuation of the lane change assistant in the recording $A2$ for the seven realistic test cases has not changed in comparison to the initial evaluation with the initial set of manually modeled test cases. The percentage of *faulty abstract situations* has remained at 2.70% (cf. Section 8.2.1.3.1).
The result of experiment E2 (cf. Fig. 8.4) show that the runtime monitoring of autonomous vehicle systems allows for more realistic and efficient simulations in the system verification of autonomous vehicle systems. Test scenarios and test cases model real-world situations more accurate if they defined based on the results of the runtime monitoring during operation.

> **Conjecture C3.** *Tests based on runtime monitoring results in realistic traffic allow to achieve a satisfiable coverage of real-world situations.*
> The runtime monitoring of autonomous vehicle systems in the real world provides data for the definition of test scenarios and test cases. These test scenarios and test cases model real-world situations more accurate than requirements based test scenarios and test cases. The impact of the more realistic test scenarios and test cases have been reflected by the increased coverage of encountered traffic situations for the recording on the German highway A2 in the case study (cf. Fig. 8.4).

> The histogram in Fig. 8.4a suggests that a satisfiable coverage of real-world situations can be achieved by system verification in realistic simulations because the occurrence rate of new unique situations in the histogram slows down over time. However, this may only be valid for driving on highways. The number of possible situations in the real world is vast — if not infinite — (cf. Section 3.8.1) and other domains, e.g., urban roads, are likely to introduce new unknown traffic situations. For efficient verification and validation, these crucial and unknown situations have to be identified. There runtime monitoring in simulations of the system verification and during operation can help to find and verify these crucial situations more efficient than requirements based tests.

The results of the case study are summarized and assessed in the following section.

8.3. Summary and Assessment of the Case Study

The case study on a lane change assistant as part of a highway pilot (cf. Chapter 8) supports the validity and impact of the proposed engineering approach for the development and operation of autonomous vehicle systems. A complete round-trip has been exemplarily performed in the case study The round-trip includes (cf. Section 4.3):

- the verification of the lane change assistant in simulations based on manually modeled test cases,
- the runtime monitoring during operation in simulations of virtual traffic and recordings of real-world test drives, and
- the definition and evaluation of additional realistic test scenarios and test cases from recordings by the runtime monitoring in the recordings of real-world test drives.

The evaluation of the runtime monitoring results in simulations and for the recordings of test drives in the case study feature the possibility to evaluate the correctness of the abstractions in the runtime monitoring framework (cf. Conjecture **C1**). The correctness of the *input abstraction* of the runtime monitoring framework can be estimated based on the percentage of *tested* and *incorrect abstract situations*. Any percentage of *tested* and *incorrect abstract situations* than 0% indicates faults in the definition and/or implementation of the runtime monitoring framework.
The first experiment in the case study discloses that manually modeled test cases from system requirements are unlikely to realistically model the traffic situations which autonomous vehicle systems will encounter in the real world (cf. Conjecture **C2**). Test engineers tend to model the most straightforward test scenarios and test cases for the verification of the addressed system requirements but do not incorporate real-world traffic. The test scenarios and test cases modeled from the recording of *abstract situation* by the runtime monitoring in the recording of the real world test drive of the first experiment lead to significantly increased coverage of situations in these recordings.

Statistics about the encountered situations from these experiments disclose that autonomous vehicle systems encounter traffic situations in different quantities in the real world and that rate at which new situations are encountered reduces with the number of known traffic Situations. These statistics let suspect that a large number of real-world situations for the operation on highways can be verified in the system verification (cf. Conjecture **C3**).
The efficiency of the verification and validation in simulations is subject to the composition of test scenarios and test cases. First, crucial and impactful situations and scenarios have to be identified. The runtime monitoring of this work and large fleets of prototype vehicle can help to increase corresponding data and identity such situations faster significantly. Subsequently, identified situations and scenarios have to be efficiently combined.
The test suite should consist of a sufficient diverge set of test cases that cover a large space of real-world situations with a low number of tests case. Not all test cases may be as impactful in composition with other tests cases om the test suite as they are individually (cf. TC5 in Fig. 8.4b). The result of the runtime monitoring in simulations can be used as an indicator for the efficiency of the selected test suite.
The scope of this case study has been limited with its 24 initial and 7 additional test cases, and two recordings from the real-world test drives. Unfortunately, the project in which this case study has been performed was restricted by its duration and available funds. An extensive case study with a large fleet of prototype vehicles would be desirable to accumulate large quantities of real-world data for an exhaustive evaluation of the engineering approach and to confirm the results of this case study. Furthermore, the real prototype vehicle would enable to consider the implementation of the runtime monitoring framework in real embedded systems without the usage of ADTF.
Nevertheless, the evaluation results of this case study indicate that tests in current *state-of-the-art* simulation frameworks (at least in this evaluation) do not provide operation condition to sufficiently resemble real worlds conditions on the roads for the verification of autonomous vehicle systems. The realism in the simulations has to significantly converge towards the real world in order for their results self-evidently transfer to the operation of autonomous vehicle systems in the real world.
Amongst other things, the results of this evaluation are taken into account in the discussion of the engineering approach and the assessment of the engineering approach in the following section.

9. Conclusion

The previous chapters have extensively described the proposed engineering approach for the integration of system development and system operation of autonomous vehicle systems. This chapter summarizes the content of this thesis, discuss the engineering approach and provides an outlook on future work.

9.1. Summary

Autonomous vehicle systems introduce new challenges for the development process in the automotive domain. The analysis in Chapter 3 discloses several shortcomings in the current development process for the environment perception and modeling, decision making, and verification and validation of autonomous vehicle systems.
This thesis proposes an engineering approach as an extension of the current development process by integrating runtime monitoring of autonomous vehicle systems in simulations of the system verification and during operation in the real world in order to continuously improve the safety of autonomous vehicle systems (cf. Chapter 4). A runtime monitoring framework verifies the correctness and safety of the autonomous vehicle systems and evaluates if encountered traffic situations have been verified in simulations of the system verification. Traffic situations with incorrect and unsafe system behavior and unknown and unverified traffic situations are recorded for system improvements.
A architecture is defined for the *runtime* monitoring consisting of 5 layers (cf. Chapter 5). The *system layer* partitions the autonomous vehicle system into *preprocessing*, the monitored *function*, and *postprocessing*. The interfaces between these parts are used by the runtime monitoring to access the input and output data of the *function*.
In simulations, the preprocessing and postprocessing of autonomous vehicle systems are partially or fully substituted by simulation frameworks. The runtime data from the autonomous vehicle systems is abstracted by the *input abstraction* and *output abstraction* of the *abstraction layer*. Traffic situations as input for the *function* are transformed into *abstract situations*. Based on these *abstract situations*, the *abstract function* and *conformity oracle* of the *qualitative monitoring layer* verify the correctness and safety of the autonomous vehicle systems and *situation monitor* and *tested situation knoweldge* of the *quantitative monitoring layer* evaluate the previous verification of *abstract situations* in simulations.
The abstraction of the runtime monitoring framework is individually defined based on the requirements of each autonomous vehicle system (cf. Chapter 6). The system requirements are transformed into a typed first-order logic in a pattern-based analysis. The wording of requirements in the specific pattern allows the identification of objects,

functions, and predicates for the typed first-order logic. The typed first-order logic is considered in the definition and implementation of runtime monitoring components. Object, functions, and predicates are included in the definition and implementation of the *abstraction layer*. Predicates and formulas are considered for the implementation of the *qualitative monitoring layer*.
The engineering approach uses the runtime monitoring framework identically in simulations of the system verification and during operation of autonomous vehicle systems in the real world. In simulations of the system verification (cf. Chapter 6), the *qualitative monitoring layer* evaluates the behavior of the autonomous vehicle systems as test oracle in addition to the manual evaluation by test engineers. The *quantitative monitoring layer* records all encountered *abstract situation* in these simulations.
During operation in the real world, the *qualitative monitoring layer* evaluates the behavior of the autonomous vehicle systems as in the simulations of the system verification. The *quantitative monitoring layer* compares encountered *abstract situation* during operation with the *abstract situation* from the simulations in the *tested situation knowledge* in order to evaluate if the autonomous vehicle system has already been verified in the encountered situations. The runtime monitoring during operation allows to initiate safety measures in unknown situations and situations with incorrect and unsafe system behavior (cf. Section 7.1.1) and to record these situations for improvement of the autonomous vehicle systems and their simulations (cf. Section 7.1.2). This thesis has not addressed the selection and execution.
Recorded *abstract situation* enable system engineers to analyze and improve autonomous vehicle systems in additional development iterations by locating and resolving insufficient system functionality. Test scenarios and test cases are defined for the system verification based on the recorded situations. Test scenarios model the behavior of dynamic objects and changes of the scenery on the abstract level of the system requirements. For simulations in the system verification, test scenarios are transformed into test cases. These test cases define concrete candidates for abstract parameters of the test scenarios. The integration of simulations and operation in the real world by runtime monitoring enable to iteratively extend the scope of safety assurance in the development of autonomous vehicle systems.
The engineering approach has been evaluated in a case study on a lane change assistant of an industrial highway pilot (cf. Chapter 8). In a first experiment, the lane change assistant has been evaluated for a set of test cases which have been manually modeled based on the requirements of the lane change assistant. Afterward, the behavior of the lane change assistant has been monitored in simulations of virtual test tracks with random traffic and recordings from real-world test drives with the recorded situations from the simulations of the manually modeled test cases. The results by the runtime monitoring in this first experiment show that the manually modeled test cases fail to sufficiently address traffic situations in the real world.
As a consequence of the first experiment, seven additional test cases have been manually modeled and simulated based on the runtime monitoring results from the recordings of the real world test drives. In the reevaluation of the runtime monitoring results for these real-world recordings, the *tested situation knowledge* from the seven additional

real-world test cases significantly improved the coverage of encountered traffic situations in comparison to the initial manually modeled test cases.

The following section discusses the impact of the proposed engineering approach for the development of autonomous vehicle systems in more detail.

9.2. Discussion

The proposed engineering approach and runtime monitoring framework are envisaged to support the development of safe autonomous vehicles. The following discussion assesses the impact of the engineering approach for the development of autonomous vehicle systems.

> ***Research Goal:*** *Enhancement of the current development practice for autonomous vehicle systems into a holistic engineering approach supporting the qualitative and quantitative supervision and estimation of correctness and safety by the seamless integration of system development and system operation.*
>
> The research goal of this thesis is addressed by the proposed engineering approach and its runtime monitoring in simulations and during operation in the real world.

The engineering approach is the first approach to integrate runtime monitoring in simulations and during operation in the real world into a holistic approach (cf. research goal). This holistic engineering approach enables more efficient development of autonomous vehicle systems with an increased level of safety. Other approaches (cf. [Bac+15; Bac+17a; Bac+17b; Zof+15]) only envisage the use of runtime monitoring from the operation in the real world for simulations of the system verification but do not address the validation of simulation results in the real world.

Car manufacturers require no significant changes to their current development process for the application of the engineering approach in the development of autonomous vehicle systems because the engineering approach is an extension of the currently prevailing development process — the v-model (cf. [SZ13]) — in the automotive domain. The complete development of autonomous vehicle systems can benefit from recorded real-world data by the runtime monitoring during operation. The most significant impact by the engineering approach is on the system verification, validation, and operation in the real world.

A runtime monitoring framework is introduced whose architecture enables the evaluation of system behavior and its safe limits for autonomous vehicle systems in simulations and during operation in the real world (cf. Chapter 5). Data about system behavior and traffic situations can be transferred between system development and system operation if the autonomous vehicle systems and their data interfaces to the runtime monitoring framework remain unchanged between simulations and operation in the real world (cf. research question 1).

Research question 1: *Which are the necessary (technical) foundations for the integration of system development and system operation in order to qualitatively and quantitatively supervise and estimate the safety of autonomous system operating in the real world?*

The technical foundation for the integration of system development and system operation is directly addressed by the architecture of the runtime monitoring framework (cf. Chapter 5). *System layer* and *simulation layer* define the integration of runtime monitoring framework and autonomous vehicle systems in simulation and during operation in the real world.

The runtime monitoring evaluates autonomous vehicle systems, their behavior, and their safe limits in traffic situations on an abstract semantic level. This semantic level abstracts from the characteristics of the internal data in autonomous vehicle systems and the real world. The abstract semantic interpretation consists of qualitative properties which are systemically derived from system requirements (cf. research question 2). In contrast to most other runtime monitoring approaches, the semantic interpretation of the runtime monitoring explicitly considers and monitors environmental situations.

Research question 2: *How to identify and define qualitative properties for each autonomous vehicle system that sufficiently represent its correctness and safety?*

In the proposed engineering approach, qualitative properties for the runtime monitoring of autonomous vehicle systems are defined based on the system requirements of these systems. The usage of system requirements for the definition of qualitative properties enables the runtime monitoring in simulations and during operation in the real world to monitor the behavior of autonomous vehicle systems and their environments on an abstract semantic level instead of internal system signals.
The runtime monitoring is only able to verify the behavior of autonomous vehicle systems for traffic situations which are sufficiently defined in the system requirements. The complete and accurate description of system behavior and analogous traffic situations by the requirements of autonomous vehicle systems are subject to the result of the requirements engineering in the development of these systems.

The inherent semantic abstraction of the runtime monitoring is related to the way engineers as well as the general public think about the behavior of autonomous vehicle systems. Therefore, the runtime monitoring could be able to help engineers and non-technical person to understand better the behavior of autonomous vehicle system in order to raise the general acceptance of this technology (cf. [SSS17]).

The runtime monitoring in simulations and during operation in the real world enables the transfer of data about the system behavior and environment to be shared between simulations and operation in the real world. The qualitative properties of autonomous vehicle systems can be monitored through their complete life-cycles — from requirements analysis to system operation in the real world (cf. research question 3)

Research question 3: *How to monitor the qualitative properties and their scopes throughout the complete life-cycle of autonomous vehicle systems — from system specification, design, implementation, and verification to operation in the real world?*

The integration of system development and system operation by the engineering approach enables the *qualitative* and *quantitative runtime monitoring* to trace and evaluate *qualitative properties* and their *quantitative scopes* through all activities of the system life cycle. The considerations of qualitative properties throughout the system development manifest themselves in the system implementation. Therefore, it is sufficient to monitor the *qualitative properties* and their *quantitative scopes* for the system implementation in simulations of the system verification and during operation in the real world.

Runtime monitoring results from simulations of the system verification are incorporated in the runtime monitoring of autonomous vehicle systems during operation in the real world. The transfer of runtime monitoring results enables estimations about the realism of simulations and the validity of their results for operation in the real world as well as the identification and evaluation of verified limits for safe behavior of autonomous vehicle systems during operation in the real world (cf. [Kna+17a]).
The distinction between safe and critical traffic situations by the *quantitative runtime monitoring* during operation allows for estimations about the residual risks by autonomous vehicle systems during operation in the real world (cf. research question 4). Traffic situations which have not been verified in the simulation or which exhibit faulty system behavior are identified and record by the runtime monitoring framework for analysis in further development iterations of the autonomous vehicle systems (cf. research question 5).

Research question 4: *How to estimate the residual risks of autonomous vehicle systems during operation in the real world based on the qualitative and quantitative knowledge from their development?*

The transfers of *qualitative properties* and their *quantitative scopes* from simulations of the system verification to the runtime monitoring during operation in the real world enables the runtime monitoring during operation to identify residual risks for autonomous vehicle systems during operation in the real world based on unknown and unsafe *abstract situation*. Safety measures can be initiated upon identification of residual risks online during operation in order to immediately mitigate these emerging residual risks.

The consideration of runtime monitoring results from the operation in the real world in the system development impacts The quality of autonomous vehicle systems and the efficiency of their development. Results of the runtime monitoring in simulations of the system verification are used by the runtime monitoring of autonomous vehicle system during operation in the real world in order to record solely critical traffic situations and dismiss known and verified traffic situations. This selective recording during operation leads to significantly more efficient identification and improvement of system faults and insufficient system behavior in the following development iteration of autonomous vehicle

systems. System engineers are not required to identify and extract incorrect system behavior and critical traffic situations from log files of arbitrary real-world data which include safe and unsafe system behavior in mixed-critical situations.

> ***Research question 5:*** *How to record violations of qualitative properties and their scopes during operation in the real world for further analysis and system improvements?*
>
> The knowledge of verified *abstract situation* from simulations in the system verification enables the runtime monitoring during operation in the real world to record *abstract situation* which have not yet been verified or which exhibit unsafe system behavior. The recordings are used in additional development iterations for improvements of autonomous vehicle systems.

The runtime monitoring data from the operation in the real world further allow the extension of safe limits for autonomous vehicle system in simulations of the system verification. Additional test scenarios and test cases can be systematically defined from recorded real-world traffic situations. These test scenarios and test cases model the real world and their traffic situations more precisely. In the case study on a lane change assistant (cf. Chapter 8), test scenarios and test cases from real-world recordings increase the coverage of encountered traffic situations to 10.37% from prior 0.68% by the manually modeled test scenarios and test cases from the system requirements of the lane change assistant (cf. Section 8.2.2).
The engineering approach represents an improvement for the current development practice of autonomous vehicle systems. Nevertheless, the approach still provides possibilities for future improvements and additional research. The following section gives an outlook on possible future research.

9.3. Future Work

The previous discussion has outlined the positive impact of the engineering approach for the development of safe autonomous vehicles. Nevertheless, the engineering approach and its implementation still offer possibilities for extensions in future work.

9.3.1. Automation of Engineering Approach

The engineering approach will only be applicable in an industrial context if a high level of automation can be achieved. The engineering approach must seamlessly integrate with the current development processes of car manufacturers and tech companies for their autonomous vehicle systems. These companies must not be forced to implement significant adjustments to their current development process for the application of the engineering approach.
The implementation of the runtime monitoring framework requires automation of the formalization of system requirements and the generation of monitoring code for various target hardware platforms. The requirement formalization could be supported by DSLs

(cf. [Mar10; VKV00]) and natural language processing (NLP) (cf. [Cho05; RP92]) while the monitor implementation could benefit from template-based code generation (cf. [Bud+96; SLS18]). Even machine learning approaches could be utilized for the automatic derivation of the abstraction in the runtime monitoring framework (cf. [LF97; Sun+15]). The code generation of the runtime monitors has to explicitly consider the embedded vehicle hardware in production vehicles, e.g., ECUs and communication buses, and the restriction of their limited resources. It might be beneficial to introduce dedicated hardware for the runtime monitoring as off-the-shelf-solution for the runtime monitoring in prototype vehicles without possible side effects on the autonomous vehicle systems. Future work should investigate the use of different variants of an autonomous vehicle system as well as entirely different autonomous vehicle systems throughout the engineering approach — in simulations and during operation in the real world as well as over multiple iterations.
The definition of test scenarios and test cases for simulation of the system verification requires the automation of the definition of test scenarios from logged traces of *abstract situation*, the derivation of test cases, and the modeling and execution of test cases in various simulation frameworks. For realistic and efficient test cases, the derivation of test cases from test scenarios should incorporate automatic fuzzing of *abstract situation* parameters (cf. [Ana+13; Bac+17c; Kha+17; Sip+16]).

9.3.2. Extending the Scope of the Runtime Monitoring

The modularity of the runtime monitoring framework allows for several extensions.
In the automotive domain, system requirements for vehicle systems frequently incorporate timing-related properties in requirements, e.g., the duration between system events or environments changes. However, the runtime monitoring in the engineering approach only monitors the autonomous vehicle systems based on system states and states of the environment at specific time stamps. Future work should incorporate timing-related properties into the runtime monitoring by using, e.g., temporal logics in the definition and implementation of runtime monitors (cf. [BKM10; Cim+10; KM08]). The consideration of time-related properties in the runtime monitoring would enable the monitoring to evaluate the behavior of autonomous vehicle systems for traffic scenarios in addition to traffic situations.
The implementation of the runtime monitoring framework in the case study solely records *abstract situation* based on the evaluation of system requirements for the behavior of autonomous vehicle systems and their environments. The modular architecture of the runtime monitoring allows for extensions in recording additional information about autonomous vehicle systems and their environments, e.g., sensors raw data. Future work should identify beneficial information about systems and environments for the development of autonomous vehicle systems and develop the functional and technical extensions to the runtime monitoring framework for recording this information.
The runtime monitoring of autonomous vehicle systems during operation in the real world enables the initiation of safety measure during operation for the mitigation of emerging safety risks by faulty system behavior or unknown situations. The selection and

execution of safety measures have not been addressed in this thesis due to the complexity in selecting safe and efficient safety measures in critical situations. For a comprehensive safety strategy, future work should research safe and sound strategies for the selection and execution of safety measures and integrated them into the runtime monitoring framework. The research should consider existing research in this field (cf. [Hör11; RM15]).

9.3.3. Improving Verification and Validation

The engineering approach depends on the quality and accuracy of simulations and the diversity and extent of test drives in the real world. A high level of realism in simulations is inevitable for the consideration of sensors in the verification and validation of autonomous vehicle systems in simulations. However, the complexity and graphic visualization of current simulation frameworks do not sufficiently match the realism of the real world.
Simulations have to sufficiently model relevant physical properties of real sensors and real-world objects in order to verify and validate physical problems for the real sensor perception and their impacts on the autonomous vehicle systems in simulations. For example, RADAR sensors have the tendency not sufficiently to recognize objects with reflective or glossy surfaces. The realism of graphical visualizations is highly relevant for the verification and validation of video camera sensors. Simulation frameworks currently lack sufficient realistic visualization as known from current video games. The complexity and realism of simulations, their environment modeling, and the modeling of sensors have to be further improved until the realism of simulations allows for comprehensive results which are intuitively applicable to the operation of autonomous vehicle systems in the real world.
Test drives in the real world are inevitable in order to verify and validate autonomous vehicle systems in the real world and to gather operational data for the engineering approach. However, the necessary configuration of prototype vehicles and the long mileage and time for the verification and validation of specific system behavior impose high costs (cf. [Sti13; Wei+14; Win15]). These costs are significantly higher than the costs of simulations.
The runtime monitoring may support the optimization of real-world test drives and reduce corresponding costs. Results from the runtime monitoring during operation of autonomous vehicle systems on public roads can be used to analyze and classify roads and their traffic. These classifications can be used in the planning of test drives. Test drives can be planed and performed on public roads which are likely to provide novel insight about road traffic or are likely to verify and validate specific autonomous system behavior, e.g., lane changes. Future work should investigate the integration of runtime monitoring results, positioning data, map data, and machine-learning techniques to provide automatic route planning for test drives with specific test goals.

9.3.4. Comprehensive Safety Strategy

The engineering approach addresses the development, verification, and validation of software functions. Results by the runtime monitoring framework apply solely to the

monitored part of the autonomous vehicle systems — the *function* (cf. Fig. 5.1). Technical components, e.g., sensors and actuators, are currently not incorporated into the engineering approach and have to be verified and validated by other V&V methods. Nevertheless, a comprehensive safety strategy, which includes hardware components, e.g., sensors and actuators, is essential for the introduction of autonomous vehicle systems to public traffic. Future research has to extend the engineering approach to these technical components by investigating extensions for the presented runtime monitoring and by introducing novel runtime monitoring concepts.

The *functional safety cages* by [Hec+11] and the *dependability cages* in [Ani+18b] present concepts for the integration of safety functions into autonomous vehicle systems in large extent. The engineering approach and its runtime monitoring represent a potential realization of these *cages* for in-vehicle software functions with complex or learned behavior. Future work should investigate the necessary modifications of the engineering approach for the implementation of these *cage* concepts.

In the context of functional safety cages and dependability cages, the implementation of the runtime monitoring as additional safety function in production vehicles and the integration of data from production vehicles into the engineering approach should be investigated. The integration as a safety function in production vehicle imposes additional requirements for the development of runtime monitors because safety standards like the safety standard ISO 26262 have to be considered. Furthermore, the data recorded by production vehicle is not directly accessible for car manufacturers, and the data storage in production vehicles is limited in general. Novel solutions for the storage and access of runtime monitoring data have to be realized in order to integrate production vehicles into the engineering approach. The verification and validation of autonomous vehicle systems can benefit from large amounts of data recorded over the large mileages which their owners commonly operate these vehicles.

9.3.5. Application in Rural and Urban Domains

In this thesis, the engineering approach has been applied to a lane change assistant which solely operates on highways. Future research should investigate the application of the engineering approach to other autonomous vehicle systems used on rural and urban roads. The research has to evaluate the transfer of the engineering approach to rural and urban domains and identify necessary modifications and extensions. For example, the increasing complexity of rural and urban traffic by additional traffic participants, e.g., pedestrians and cyclists, and different infrastructure, e.g., traffic signals, require a revision of the *abstract situation* interpretation by the runtime monitoring.

The application of the engineering approach to other domains should include additional industrial case studies. These industrial case studies should incorporate large and diverse fleets of vehicles with various autonomous vehicle systems. The amount of data recorded by these fleets would allow reinforcing results and estimations from the case study of this thesis — especially the prediction about the contentious decrease in the discovery of new *abstract situations* over time.

The application of the engineering approach to rural and urban traffic should help to investigate the possibility of a single common situation description throughout all traffic domains. The majority of autonomous vehicle systems operate in the same environments. A standard description of traffic and the behavior of vehicles across all manufacturers and tech companies would support the standardization of safety assessment for autonomous vehicle systems as well as consistent communication about the safety of autonomous vehicle system to the general public.

9.3.6. Metric for Safety of Autonomous Vehicles

Autonomous vehicle systems have to be sufficiently safe for a positive impact on public traffic. Car manufacturers and tech companies are required by national authorities to prove the safety of their autonomous vehicle systems sufficiently. The qualification of residual risks based on failure rates is not applicable to autonomous vehicle systems because the required mileage is not feasible under reasonable costs (cf. [Win15]). Novel metrics have to be introduced for qualification of residual risks by autonomous vehicle systems in public traffic(cf. [Sip+16]).
The engineering approach can support such metrics because the runtime monitoring during operation can estimate the safe behavior limits of autonomous vehicle systems in the real world. Future work should investigate the definition of new metrics for the safety of autonomous vehicle systems. These metrics should incorporate the results about encountered safe resp. critical situations from the runtime monitoring during operation in the real world (cf. [AHR15]). Information about safe resp. critical traffic situations could allow quantifying the residual risks of autonomous vehicle based on traffic situations and not on system failures.
A metric incorporating traffic situations instead of failure rates could also be beneficial for the selection of test scenarios and test cases for the system verification of autonomous vehicle systems. Future research should investigate how the quantification of the safety impact by individual test scenarios and test case could be used for the selection of the test scenarios and test cases for the test suite of the system verification. The goal is to define an efficient test suite which verifies autonomous vehicle systems for a large number of traffic situations under reasonable time and costs.

9.3.7. General Understanding about System Safety

A safety metric quantifies the safety resp. remaining residual risks for autonomous vehicle systems. In general, the approval of autonomous vehicles to public traffic rests on the comparison of quantified safety to the required safety threshold by national authorities. However, a commonly agreed safety threshold has not yet been defined for autonomous vehicles. Public discourse about the publicly excepted impact and publicly accepted drawbacks of autonomous vehicles is required. National authorities should incorporate the results of such discourse in the definition of the legally required safety for the operation of autonomous vehicles in public traffic.

The public discourse significantly impacts the acceptance of autonomous vehicles by the general public. Clear communication and description about impacts and shortcomings of autonomous vehicles are essential for this public discourse and the public acceptance (cf. [Fra+18]). The behavior of autonomous vehicle systems has to be described and communicated by car manufacturers and tech companies in a way that non-technical people can understand the decision-making of autonomous vehicle systems and relate it to their impacts and shortcomings (cf. [SSS17]). Car manufacturers and tech companies should support an early discourse by publishing available information and data from their trials with prototype vehicles on public roads.
For the approval of autonomous vehicles and their acceptance by the general public, all results about the safety of autonomous vehicle systems in their verification and validate have to ensure high fidelity and trust. Any manipulation of verification and validation results have to be excluded. At the moment, statements about the safety of autonomous vehicle systems are subject to estimation by car manufacturers and tech companies under the usage of inappropriate verification and validation methods, e.g., failures rates in real-world test drives. In the future, independent third parties will have to validate the safety statements by car manufacturers, and tech companies or technical solutions will have to be introduced for the direct and secure integration of fidelity guarantees into the data of safety statements.
A standardized safety assessment for autonomous vehicle systems would inherently provide such fidelity guarantees and enable the comparison of systems from different manufacturers. National authorities, car manufacturers, and tech companies would have to agree on such a standardized safety assessment. This agreement would include the definition of necessary and relevant data, e.g., road topologies and traffic scenarios.
Current research projects, like the PEGASUS project (cf. [Win+18]), try to define such a V&V data set as a set of relevant traffic scenarios by manual analysis of system requirements and real-world traffic. It remains questionable if these traffic scenarios sufficiently verify the safety of autonomous vehicles. The results from the case study suggest that a diverse set of realistic traffic scenarios from real-world recordings would be the better approach (cf. Chapter 8). A database of traffic scenarios should be established that is commonly accepted by all car manufacturers, tech companies, and legal authorities for the approval of autonomous vehicles. Car manufacturers and tech companies would be required to cooperate and share their recorded data from tests with their prototype vehicles on public roads with each other in order to build a sufficiently large and diverse V&V database.

A. Appendix

A.1. Data Structures in the Case Study

The following UML diagrams present the data structures for the *abstract situation* and the abstract targets as they have been used in the case study for the runtime monitoring of the lane change assistant (cf. Chapter 8).

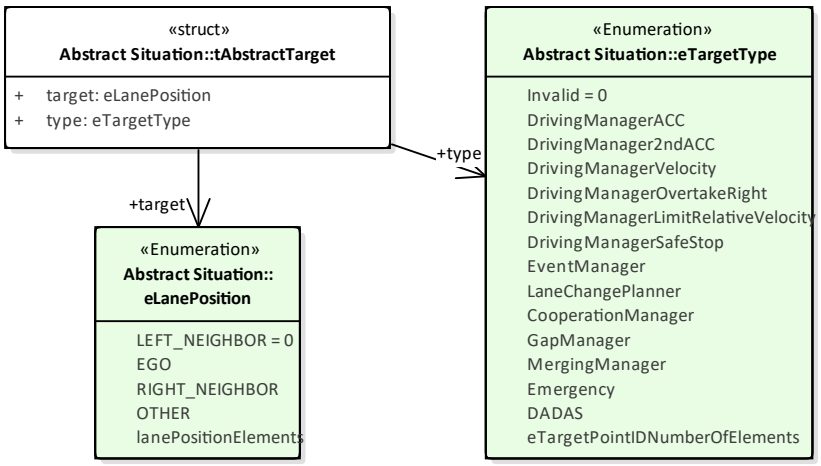


Figure A.1.: Data structure of the abstract target.

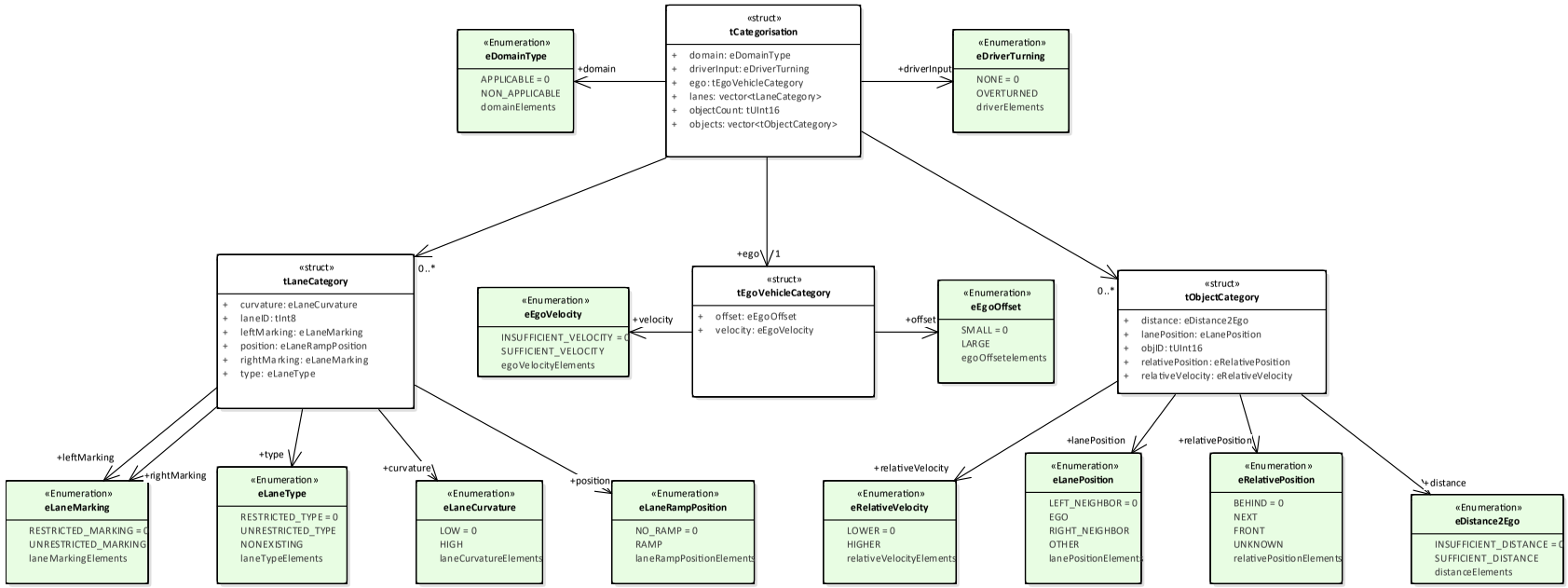


Figure A.2.: Data structure of the abstract situation.

A.2. Requirements-based Test Scenarios and Test Cases

For the verification of the lane change assistant, one of the project partners has modeled test scenarios and test cases based on the system requirements of the lane change assistant (cf. Section 3.2). The test scenarios and corresponding test cases are presented in the following tables.

Table A.1.: Test Scenario 1.

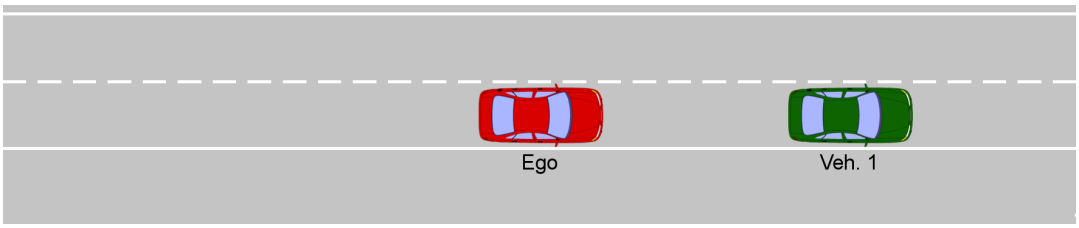
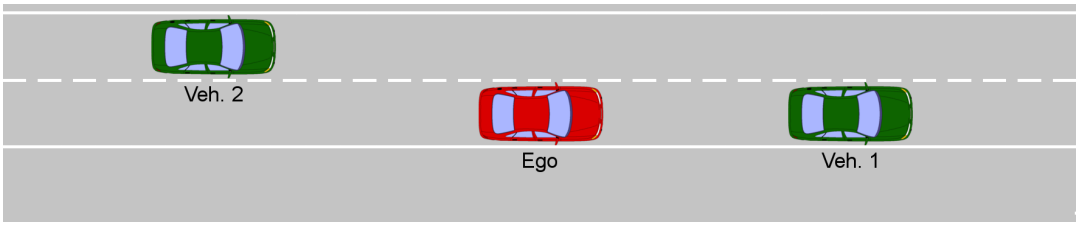
Initial Situation			
			
Requirements			
FR_1_1: The system shall be able to perform lane changes on multi-lane highways. FR_7_1_3: The system has to evaluate whether a lane change to a neighbor lane is possible based on objects on its current lane in front itself. FR_5_1: In case, the system is in the automated driving mode, the system has to determine if a lane change is beneficial based on the current traffic situation. FR_2_2: The system shall be able to prevent lane changes if the curvature of the ego lane is less than 125 m.			
Parametrization by Test Cases			
	Test Case 1	Test Case 2	Test Case 3
Road layout	2 lane highway	2 lane highway	2 lane highway
Road Radius	∞	∞	900m
Speed Limit	130 km/h	130 km/h	130 km/h
Velocity Ego	130 km/h	130 km/h	130 km/h
Velocity Veh. 1	100 km/h	100 km/h	100 km/h
Distance Veh. 1 to Ego	100 m	50 m	100 m
Expected Result	lane change (left)	remain	lane change (left)

Table A.2.: Test Scenario 2.

Scenario 2

Initial Situation



Requirements

FR_7_1_1: The system has to evaluate whether a lane change to a neighbor lane is possible based on objects on that neighbor lane behind itself.

Parametrization by Test Cases

	Test Case 4	Test Case 5	Test Case 6
Road layout	2 lane highway	2 lanes highway	2 lanes highway
Road Curvature	∞	∞	∞
Speed Limit			
Velocity Ego	130 km/h	130 km/h	130 km/h
Velocity Veh. 1	100 km/h	100 km/h	100 km/h
Distance Veh. 1 to Ego	50 km/h	50 km/h	50 km/h
Velocity Veh. 2	140 km/h	100 km/h	*No Veh. 2*
Distance Veh. 2 to Ego	−250 m	−50 m	*No Veh. 2*
Expected Result	remain	lane change (left)	remain

Table A.3.: Test Scenario 3.

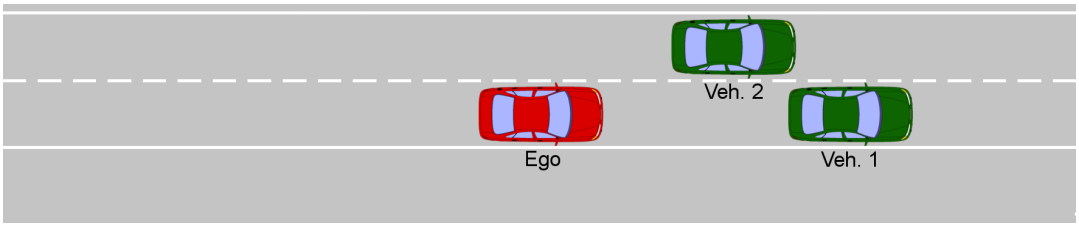
Scenario 3				
Initial Situation				
				
Requirements				
FR_7_1_2: The system has to evaluate whether a lane change to a neighbor lane is possible based on objects on that neighbor lane in front of itself. FR_7_1_5: The system shall be able to consider objects directly next to it with a relative velocity of less than 5 m/s.				
Parametrization by Test Cases				
	Test Case 7	Test Case 8	Test Case 9	Test Case 10
Road layout	2 lane highway	2 lane highway	2 lane highway	2 lane highway
Road Curvature	∞	∞	∞	∞
Speed Limit	130 km/h	130 km/h	130 km/h	130 km/h
Velocity Ego	130 km/h	130 km/h	130 km/h	130 km/h
Velocity Veh. 1	100 km/h	100 km/h	100 km/h	130 km/h
Distance Veh. 1 to Ego	50 m	50 m	50 m	50 m
Velocity Veh. 2	81 km/h	105 km/h	130 km/h	100 km/h
Distance Veh. 2 to Ego	100 m	0 m	0 m	100 m
Expected Result	Remain (decelerate) or lane change (left)	lane change (left)	remain	Remain (decelerate) or lane change (left)

Table A.4.: Test Scenario 4.

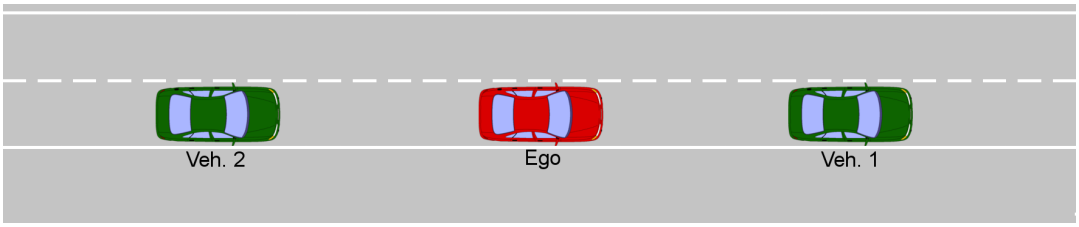
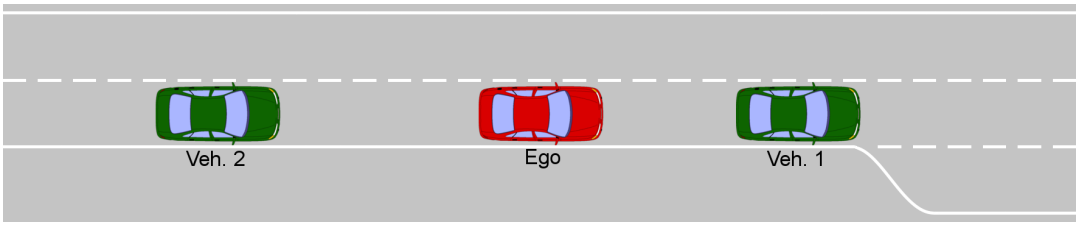
Scenario 3			
Initial Situation			
			
Requirements			
FR_7_1_4: The system shall evaluate whether a lane change is possible based on approaching objects on the ego lane behind itself. FR_2_3: The system shall be able to prevent lane changes if the lateral offset to the center of the current lane is more than 0.4 m.			
Parametrization by Test Cases			
	Test Case 11	Test Case 12	Test Case 13
Road layout	2 lane highway	2 lane highway	2 lane highway
Road Curvature	∞	∞	∞
Speed Limit	130 km/h	130 km/h	130 km/h
Velocity Ego	110 km/h	110 km/h	110 km/h
Lateral offset Ego	0 m	0 m	0.5 m
Velocity Veh. 1	100 km/h	100 km/h	100 km/h
Distance Veh. 1 to Ego	50 m	50 m	50 m
Velocity Veh. 2	100 km/h	140 km/h	100 km/h
Distance Veh. 2 to Ego	−50 m	−200 m	−50 m
Start Overtaking Veh. 2		−50 m	
Expected Result	lane change (left)	remain	remain

Table A.5.: Test Scenario 5.

Scenario 5

Initial Situation



Requirements

FR_6_1: The system shall be able to perform lane changes in onto off-ramps less than 100 m after the off-ramp started.
FR_5_1_3: The system shall be able to assess, whether a lane change is beneficial based on timing restrictions regarding prior driving events.

Parametrization by Test Cases

	Test Case 14	Test Case 15	Test Case 16	Test Case 17
Road layout	2 lane highway w. off-ramp	2 lane highway w. off-ramp	2 lane highway w. off-ramp	2 lane highway w. off-ramp
Road Curvature	∞	∞	∞	∞
Speed Limit	130 km/h	130 km/h	130 km/h	130 km/h
Velocity Ego	130 km/h	130 km/h	130 km/h	130 km/h
Ego distance to off-ramp	300 m	300 m	300 m	300 m
Mission Ego:	Follow Highway	Leave Highway	Leave Highway	Leave Highway
Velocity Veh. 1			100 km/h	110 km/h
Distance Veh. 1 to Ego	50 m	50 m	50 m	50 m
Mission Veh. 1			Follow Highway	Follow Highway
Velocity Veh. 2				100 km/h
Distance Veh. 2 to Ego				100 m
Start Overtaking Veh. 2				−50 m
Expected Result	remain	lane change (right)	lane change (right)	remain

Table A.6.: Test Scenario 6.

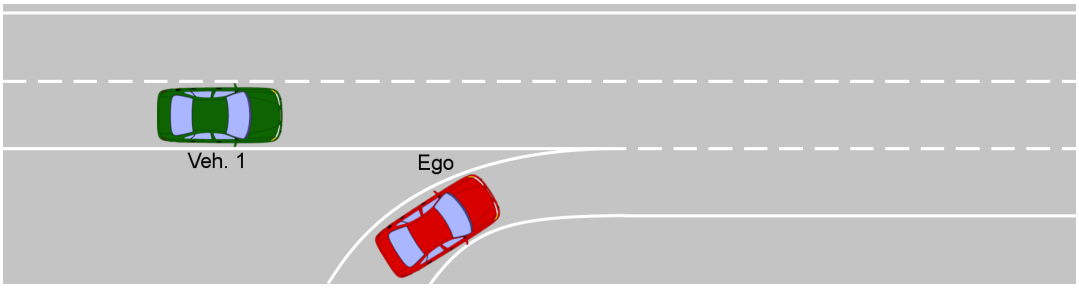
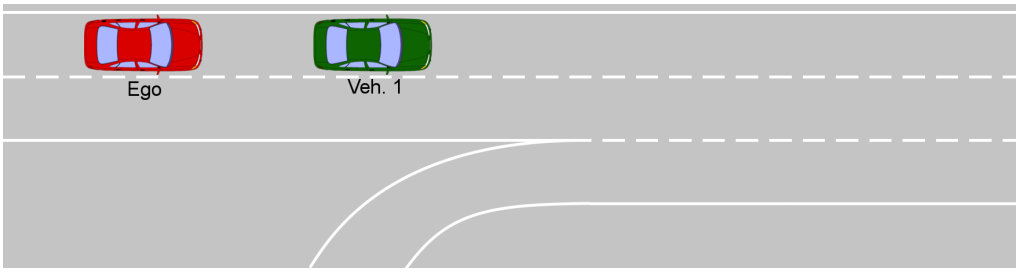
Scenario 6		
Initial Situation		
		
Requirements		
FR_6: The system shall be able to perform lane changes to enter a highway on an on-ramp.		
Parametrization by Test Cases		
	Test Case 18	Test Case 19
Road layout	2 lane highway w. on-ramp	2 lane highway w. on-ramp
Road Curvature	∞	∞
Speed Limit	130 km/h	130 km/h
Velocity Ego	100 km/h	100 km/h
Ego distance to highway	200 m	200 m
Velocity Veh. 1		100 km/h
Distance Veh. 1 to Ego		0 m
Expected Result	lane change (left)	lane change (left)

Table A.7.: Test Scenario 7.

Scenario 7

Initial Situation



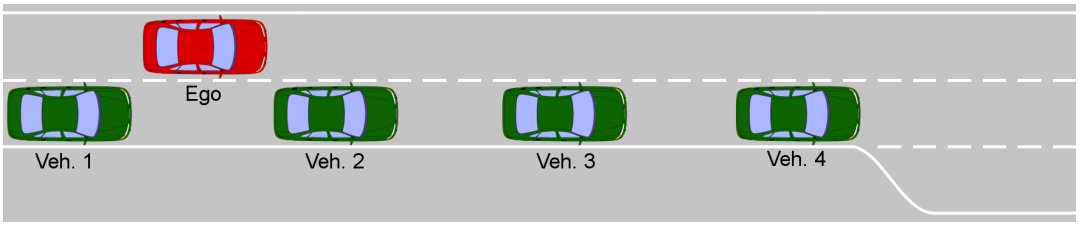
Requirements

FR_6_3: The system shall prevent lane changes towards areas with an adjacent highway on-ramp.

Parametrization by Test Cases

	Test Case 20	Test Case 21	Test Case 22
Road layout	2 lane highway w. on-ramp	2 lane highway w. on-ramp	2 lane highway w. on-ramp
Road Curvature	∞	∞	∞
Speed Limit	130 km/h	130 km/h	130 km/h
Velocity Ego	130 km/h	130 km/h	130 km/h
Ego distance to on-ramp	200 m	200 m	250 m
Velocity Veh. 1		100 km/h	150 km/h
Distance Veh. 1 to Ego		100 m	−100 m
Expected Result	remain	remain	remain

Table A.8.: Test Scenario 8.

Scenario 8		
Initial Situation		
		
Requirements		
FR_3: The system shall be able to adjust the vehicle longitudinally towards a suitable gap in a search range of 100 m to the front and 100 m to the rear of the vehicle in order to prepare a lane change. FR_9_1: The system has to evaluate whether a lane change is possible based on the lane marking type of the appropriate lane boundary.		
Parametrization by Test Cases		
	Test Case 23	Test Case 24
Road layout	2 lane highway w. off-ramp	2 lane highway w. off-ramp
Road Curvature	∞	∞
Speed Limit	130 km/h	130 km/h
Lane Marking	dashed	solid
Velocity Ego	130 km/h	130 km/h
Ego distance to off-ramp	900 m	900 m
Velocity Veh. 1	100 km/h	100 km/h
Distance Veh. 1 to Ego	−30 m	−30 m
Velocity Veh. 2	100 km/h	100 km/h
Distance Veh. 2 to Ego	0 m	0 m
Velocity Veh. 3	100 km/h	100 km/h
Distance Veh. 3 to Ego	40 m	400 m
Velocity Veh. 4	100 km/h	100 km/h
Distance Veh. 4 to Ego	100 m	100 m
Expected Result	lane change (right)	remain

A.3. Test Scenarios from Real World Recoding

The following tables present the initial situations and the list of maneuvers by dynamic objects in the vicinity of the automated vehicle with the lane change assistant for the eight test scenarios which have been modeled in the case study based on the runtime monitoring results (cf. Chapter 8). The sequences of object maneuvers in the test scenarios are relevant for the correct emergences of the envisaged *abstract situation* in the simulations. The envisioned situations correspond to the recorded *abstract situation* by the runtime monitoring in the recordings from the German Highway A2.

Table A.9.: Real World Scenario 1.

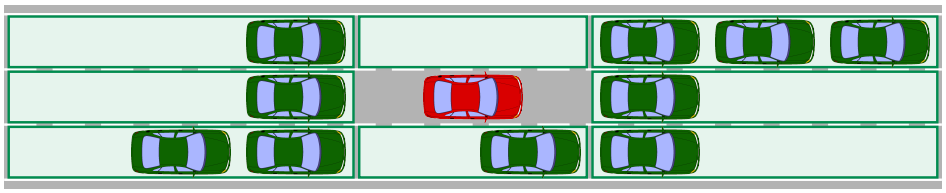
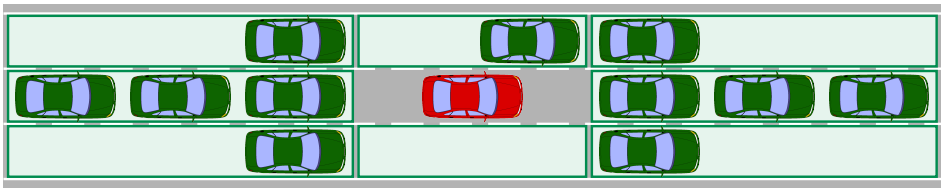
Real World Scenario 1
Initial Situation

Sequence of Maneuvers
1. Vehicle in zone *front-right* falls back into zone *next-right*.
2. Lane change of vehicle in zone *rear-right* to *rear-ego*.
3. Vehicle in zone *next-right* drives up to zone *front-right*.
4. Lane change of vehicle in zone *front-left* to zone *front-ego*.
5. Lane change of vehicle in zone *front-ego* to zone *front-left*.
6. Vehicle in zone *front-right* falls back into zone *next-right*.
7. Vehicle in zone *next-right* falls back into zone *rear-right*.
8. Second vehicle in zone *next-right* falls back into zone *rear-right*.
9. Vehicle in zone *rear-right* falls out of the rear sensor range.
10. Vehicle in zone *front-left* drives out of the front sensor range.
11. Vehicle in zone *rear-right* falls out of the rear sensor range.
12. Vehicle in zone *rear-ego* falls out of the rear sensor range.

Table A.10.: Real World Scenario 2.

Real World Scenario 2

Initial Situation



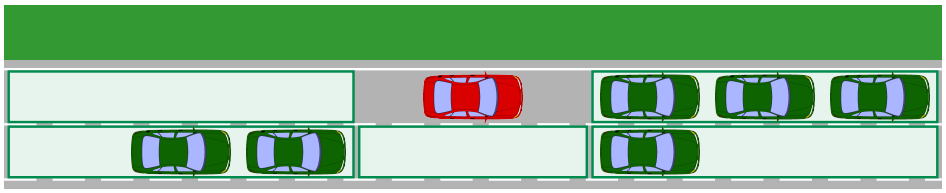
Sequence of Maneuvers

1. Lane change of vehicle in zone *front-ego* to zone *front-right*.
2. Vehicle in zone *rear-ego* falls out of the rear sensor range.
3. Vehicle falls back into zone *front-left* from outside the front sensor range.
4. Lane change of vehicle in zone *front-ego* to zone *front-right*.
5. Vehicle in zone *front-right* falls back into zone *next-right*.
6. Vehicle falls back into zone *front-right* from outside the front sensor range.
7. Vehicle in zone *front-right* falls back into zone *next-right*.
8. Lane change of vehicle in zone *rear-ego* to zone *rear-right*.
9. Vehicle drives into zone *rear-right* from outside the rear sensor range.
10. Vehicle in zone *front-right* drives out of the front sensor range.
11. Vehicle in zone *rear-right* falls out of the rear sensor range.
12. Vehicle in zone *next-right* falls back into zone *rear-right*.
13. Vehicle in zone *rear-right* falls out of the rear sensor range.
14. Vehicle in zone *next-right* falls back into zone *rear-right*.
15. Vehicle in zone *rear-right* falls out of the rear sensor range.
16. Vehicle in zone *next-left* drives up to zone *front-left*.
17. Vehicle falls back into zone *front-right* from outside the front sensor range.
18. Vehicle in zone *front-right* falls back into zone *next-right*.
19. Lane change of vehicle in zone *front-ego* to zone *front-right*.
20. Vehicle in zone *next-right* falls back into zone *rear-right*.

Table A.11.: Real World Scenario 3.

Real World Scenario 3

Initial Situation



Sequence of Maneuvers

1. Lane change of vehicle in zone *front-ego* to *front-right*.
2. Lane change of vehicle in zone *front-right* to *front-ego*.
3. Lane change of vehicle in zone *front-ego* to *front-right*.
4. Vehicle falls back into zone *front-right* from outside the front sensor range.
5. Vehicle in zone *front-ego* drives out of the front sensor range.
6. Vehicle in zone *rear-right* falls out of the rear sensor range.
7. Lane change of vehicle in zone *rear-right* to *rear-ego*.
8. Lane change of vehicle into zone *rear-right* from adjacent outside lane.
9. Lane change of vehicle into zone *next-right* from adjacent outside lane.
10. Vehicle in zone *next-right* falls back into zone *rear-right*.
11. Vehicle in zone *front-right* drives out of the front sensor range.
12. Vehicle in zone *next-right* falls back into zone *rear-right*.
13. Lane change of vehicle in zone *front-ego* to *front-right*.
14. Vehicle drives into zone *rear-ego* from outside the rear sensor range.

Table A.12.: Real World Scenario 4.

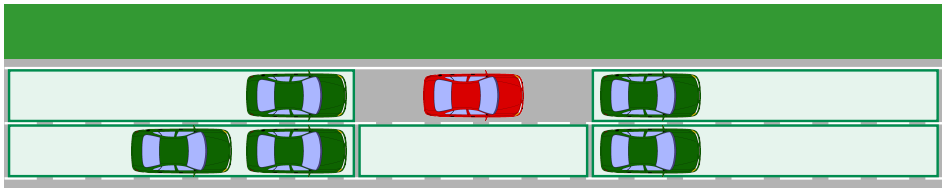
Real World Scenario 4
Initial Situation

Sequence of Maneuvers
1. Vehicle falls back into zone *front-ego* from outside the front sensor range. 2. Lane change of vehicle in zone *front-ego* to *front-right*. 3. Lane change of vehicle in zone *front-right* to adjacent outside lane.

Table A.13.: Real World Scenario 5.

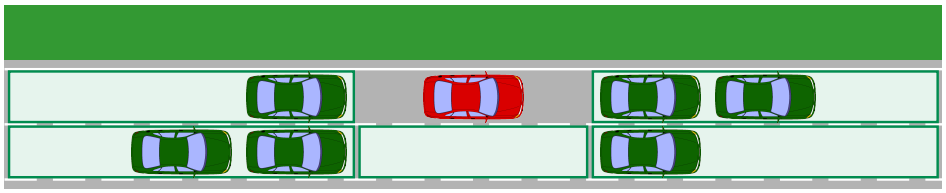
Real World Scenario 5
Initial Situation

Sequence of Maneuvers
1. Lane change of vehicle in zone *front-ego* to *front-right*. 2. Lane change of vehicle in zone *front-right* to adjacent outside lane. 3. Vehicle in zone *rear-right* falls out of the rear sensor range. 4. Vehicle in zone *rear-ego* falls out of the rear sensor range.

Table A.14.: Real World Scenario 6.

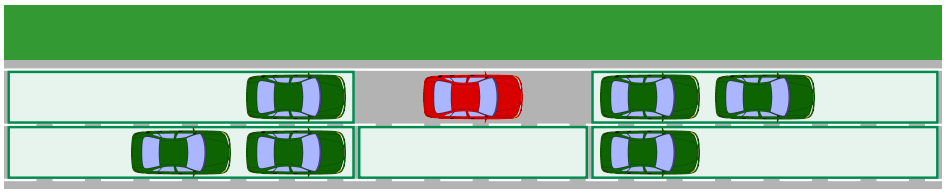
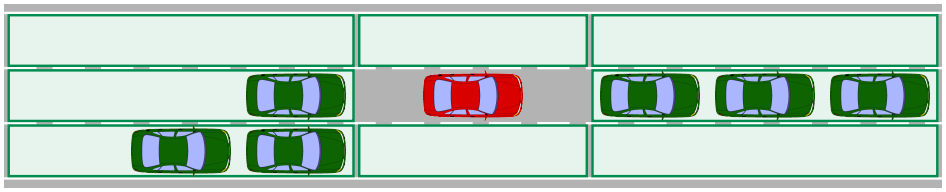
Real World Scenario 6
Initial Situation

Sequence of Maneuvers
1. Lane change of vehicle in zone *front-ego* to *front-right*. 2. Vehicle in zone *front-right* drives out of the front sensor range. 3. Lane change of vehicle in zone *rear-right* to *rear-ego*. 4. Lane change of vehicle in zone *rear-right* to adjacent outside lane. 5. Vehicle in zone *rear-ego* falls out of the rear sensor range. 6. Vehicle drives into zone *rear-right* from outside the rear sensor range. 7. Lane change of vehicle into zone *front-right* from adjacent outside lane.

Table A.15.: Real World Scenario 7.

Real World Scenario 7
Initial Situation

Sequence of Maneuvers
1. Vehicle in zone *front-ego* drives out of the front sensor range. 2. Vehicle falls back into zone *front-right* from outside the front sensor range. 3. Lane change of *Automated (ego) vehicle* to the right neighbor lane. 4. Lane change of vehicle in zone *front-left* to adjacent outside lane.

Glossary

absolute global reference frame

The positioning of objects in the real world relative to a fixed reference frame.

abstract action

The representation of the system output in the runtime monitoring framework.

abstract behavior

The behavior of a system in the abstract representation of the runtime monitoring framework. (see abstract action.

abstract representation

The representation of the system and its environment by the runtime monitoring framework.

abstract situation

The abstract representation of the system state and environment state within the runtime monitoring framework.

artifact under test

The development artifact, e.g., models, software, hardware, or system, that is being verified for correct operation in the test.

augmented reality

The overlay of computer-generated perceptual information and objects in the humans' visual, auditory, haptic, somatosensory, or olfactory perception.

behavior planning

The planning of vehicle actions from the information about the vehicle environment from the situation assessment.

black-box

The consideration of a system based on its external observable behavior without any information about the internal structure and processing of the system.

certifiability

The ability of a system to be evaluated and assessed by qualified assessors.

closed formula

A formula without any free variables.

closed-loop

The output of a system is considered for its control actions. The outputs are transferred back to the system as inputs via feedback loop.

closed-loop simulation

A feedback loop transfers the outputs of the system under test back via the environment simulation back to the system under test as inputs.

closed-world assumption

The assumption that the system environment *can* be completely specified in the systems development (design time).

cluster

A trace of abstract situations which have been recorded in a cohesive time frame by the runtime monitoring framework.

code instrumentation

The integration of sensor or probes into the code of the monitored system.

counter example

A path through the state space of the system and its environment as an example of the violation of an intended system property.

decomposition

The partitioning of requirements, systems, or system components into smaller parts - sub-requirements, subsystems, or subcomponents.

dependability

The ability of a system to provide its functionality correctly for a given time-period.

design time

The time in which the system is specified, design, implmented, verified and validated.

development part

The part of the engineering approach which is concerned with the specification, design, implementation, and verification of autonomous vehicle systems.

domain

The set of (real world) objects for the semantic evaluation of first-order logic.

domain control unit

A more powerful control unit than ECU for the execution of computation-heavy and communication-heavy software functions.

domain specific language

A (formal) language which is tailored to the requirements and semantics of a particular domain.

dummy object

see mock object.

dynamic object

An object in the environment which *can* change its state for each time stamp.

electronic control unit

Embedded hardware in automotive electronics which executes software for control of one or more of vehicle systems, e.g., the engine or brakes.

environment

The set of all objects, which reside in the vicinity of a system or vehicle.

environment model

The internal representation of the vehicle environment within autonomous vehicle systems.

environment perception

The components which are involved in the perception of the vehicle environment and the definition of an internal representation of environmental situations.

environment situation

The situation in the environment of a system. For autonomous vehicles relates to traffic situation.

environment-referenced view

The environment is described in relation to the absolute global reference frame.

error

"A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition" [Int09a].

exteroceptive

That responds to external stimuli.

failure

"The termination of the ability of an element or an item to perform a function as required" [Int09a].

false negative

The *nonconformity* of a *negative* test result for a (system) property for the manifestation of the property in the real world.

false positive

The *nonconformity* of a *positive* test result for a (system) property for the manifestation of the property in the real world.

fault

"An abnormal condition that can cause an element or an item to fail" [Int09a].

feedback loop

The transfer of system outputs of back to the system as inputs.

field operational test

The test of the correct operation of a vehicle in public traffic or on test tracks.

field test

see field operational test.

field testing

The activity to verify the operation of a system in field operational test.

functionality

The ability of a system to provide its intended functionality.

fuzzing

The derivation of value for a set of concrete parameter values from a set parameters in an abstract representation. Here, the definition of concrete parameters in a test case from the abstract parameters in test scenarios.

fuzzy Parameter

A parameter which does not contain a single absolute value but an interval of possible values.

gateway

A hardware component or ECU that connects and organizes the data transfer between multiple communication systems, e.g., CAN buses.

ghost object

A virtual object as a representation of emerging objects and their maneuvers in the definition of test scenarios.

ground term

A term without any variables.

ground truth

The independent confirmation [validation] at a site for information or results obtained by remote sensing, evaluation, verification, or testing.

human machine interface

The interfaces of the system for interactions between humans and the system.

interventions

An action by a human driver to interfere with the autonomous operation of a vehicle to take over the control of the car - predominantly in critical and unsafe traffic situations.

localization

The position estimation of the ego vehicle in the (real) world.

malfunction

"An intermittent irregularity in the fulfillment of a system's desired function" [Ise06b].

maneuver

An action by a dynamic object.

mock object

A simulated object which mimics the behavior of real system objects in controlled ways.

multiple points of failure

The parts of a system which have to fail simultaneously for the entire system to stop working.

multitudinous environments

An environment with a vast number of characteristics and situations which is difficult to be precisely specified.

natural language processing

The computer-based processing and analysis of large amounts of natural language data.

object

An atomic entity of the environment or a combination of atomic entities.

object tracking

The identification and pursuit of specific objects in the data from vehicle sensor over time.

off-the-shelf

Existing Solutions or products of suppliers and vendors which are adapted to the needs of the purchasing organization rather than the commissioning of custom-made solutions or products.

offline monitoring

The verification of formally specified conditions for a system at design time on information recorded during the system operation.

online monitoring

The verification of formally specified conditions for a system during its operation at runtime (cf. runtime verification).

open-loop

The output of a system does not affect the system's control actions.

open-loop simulation

Outputs from the system under test are not considered for the simulation of the environment. The feedback loop between the SUT and environment is not closed.

open-world assumption

The assumption that the system environment *cannot* be completely specified in the systems development (design time).

operation

The method by which a system performs its function - predominantly in the real world.

operation condition

The state of system environment for the execution of the system in a specific operation mode.

operation mode

Specific states of software functions, hardware components, and the environment of a system for the execution of a particular operation or functionality by the system at runtime.

operation part

The part of the engineering approach which is concerned with the operation of autonomous vehicle systems in the real world.

parametrization

see fuzzing

powertrain

The main components of a vehicle for the generation of power and its delivery to the road surface, i.e., engine, transmission, drive shafts, differentials, and the final drive.

processing chain

The chain of the functional components in the autonomous vehicle system for the processing of actions based on the data from the environment perception.

proprioceptive

Something pertaining to the sense of the position of parts of the body, relative to other neighboring parts of a body.

rapid prototyping

Techniques for the modeling or emulation of physical systems or components for the fast execution and evaluation in the real world while the actual system or component is still developed.

ring buffer

A first-in-first-out buffer which the oldest elements is discarded if the buffer is full and a new item is added to the buffer.

road graph

The partitioning of requirements, systems, or system components into smaller parts - sub-requirements, subsystems, or subcomponents.

road safety

Absence of unreasonable risk for traffic participants.

runtime

The time during which the system is running.

runtime data

see system data.

runtime environment

The framework for the execution of software, hardware, or system; The surrounding entities of software, hardware, or system during its operation.

runtime monitor

The software and hardware components for the observation of the system execution and proving the correctness of the system execution for a formally defined specification or property.

runtime verification

The verification of formally specified conditions based on observed system properties from the monitoring of software execution (cf. [HG08]).

safe state

A state of the system in which any risks for the safety of any humans and objects are excluded.

safety

Absence of unreasonable risks.

safety measure

An action by a system or the system's safety function (in the presence of system faults or unsafe system behavior) to mitigate emerging risks for the safety of persons and objects.

scene

The configuration of the vehicle's environment as a spatial-temporal arrangement from an observers point of view - including the *scenery*, dynamic objects, and self-representation.

scenery

The set of static objects in the environment of a vehicle.

schedulability

The ability of a system process to meet their deadlines to provide its functionality correctly.

self-adaptive system

"Self-adaptive software is capable of evaluating and changing its behavior, whenever the evaluation shows that the software is not accomplishing what it was intended to do, or when better functionality or performance may be possible" [Mac+13].

simulation environment

see simulation framework.

simulation framework

A framework which integrates system under test into a co-simulation with various models for the different aspects of the system's environment.

simulation-based testing

Simulation-based testing is testing of systems using models and simulations.

single point of failure

A part of a system that will stop the entire system from working if it fails.

situation

The entirety of circumstances which results in a certain behavior of a system.

situation assessment

The assessment and augmentation of the environment representation from the environment perception with additional information for the processing of each function.

situation parameter

A parameter (dimension) in abstract situation of the runtime monitoring framework.

slicing

The partition of long test scenarios into specialized sub-scenarios with smaller durations.

specification

The collection of functional and non-functional requirements posted on a system, subsystems, or a single component.

stakeholder

A person or group with interest in the outcome of a project, activity, or decision.

state

The entirety of circumstances for all properties of an entity, object, or system.

static object

An object in the environment which *cannot* change its state.

system data

The information data which is generated by a system at operation.

system development

The specification, design, implementation, and verification of a system.

system environment

see environment.

system operation

The operation of a system in the real world.

system specification

see specification.

system state

The joint condition of the system. Here, the system denotes an autonomous vehicle system *and* its environment.

system test

Test of the complete product (system) instead of individual system components or subsystems (cf. test).

system under test

The system in a test that is being tested for correct operation (cf. artifact under test).

target point

A 2-dimensional or 3-dimensional point in front of the vehicle as the target for the calculations of the future vehicle trajectory.

test

The verification of an artifact, e.g., methods, class, components, or system, for a set of test cases.

test case

The specification of the inputs, execution conditions, testing procedure, and expected results for the execution of system under test (SUT) in order to verify if the system meets its requirements or specification

test coverage

The extent of an system that is verified by a set of test case.

test oracle

A (software) component which process the intended reaction or output of a artifact under test in a test case.

test plan

The definition and documentation of the proceeding for the testing of an artifact - including the test suite.

test scenario

A (timed) sequence of scenery changes and maneuvers by dynamic objects in the vicinity of the system under test.

test suite

The set of test cases for verifying the correct operation of an artifact, e.g., methods, class, components, or system.

testing

Testing is the process of executing a program with the intent of finding errors.

time-to-market

The time until a product is available for sale.

trace

A possibly infinite sequence of system states or system actions.

traffic situation

The state of the traffic in the vicinity of a vehicle.

type

A class of objects which share similar properties.

type hierarchy

An inheritance hierarchy of types.

unique abstract situation

An instance in a set of abstract situation which has no duplicates.

use case

A typical scenario for the operation of a system - including the interaction between the system and its environment.

validation

The activity to check the compliance of artifacts, e.g., methods, class, components, or system, for its purpose.

vehicle-referenced view

The environment is described in relation to the ego vehicle.

vehicle-under-test

The vehicle in a test that is being tested for correct operation (cf. artifact under test).

verification

Proving the correctness of a system or algorithms with respect to a formally defined specification or property.

Bibliography

[aca06] acatech - Deutsche Akademie der Technikwissenschaften e.V. “Mobilität 2020. Perspektiven für den Verkehr von Morgen. Schwerpunkt: Straßen- und Schienenverkehr”. In: *acatech BERICHTET UND EMPFIEHLT* 1 (2006).

[Ack+08] Chris Ackermann et al. “Model Based Design Verification: A Monitor Based Approach”. In: *SAE Technical Paper 2008-01-0741* (2008).

[ADT14] Thomas Arts, Michele Dorigatti, and Stefano Tonetta. “Making Implicit Safety Requirements Explicit”. In: *Proceedings of the 33rd International Conference on Computer Safety, Reliability, and Security*. Vol. 8666. LNCS. Florence, Italy: Springer International Publishing Switzerland, 2014, pp. 81–92. ISBN: 978-3-319-10505-5.

[Aga+16] Venkatesh Agaram et al. “Validation and Verification of Automated Road Vehicles”. In: *Road Vehicle Automation 3*. Ed. by Gereon Meyer and Sven Beiker. Vol. 3. Lecture Notes in Mobility. Springer International Publishing, July 2016, pp. 201–210.

[Agg14] Charu C. Aggarwal. *Data Classification: Algorithms and Applications*. Ed. by Vipin Kumar. Vol. 35. Data Mining and Knowledge Discovery Series. Chapman & Hall/CRC, 2014. ISBN: 9781466586741.

[Ahl+05] Jaswinder Ahluwalia et al. “Model-Based Run-Time Monitoring of End-to-End Deadlines”. In: *Proceedings of the 5th ACM international conference on Embedded software*. ACM. New York, NY, USA, 2005, pp. 100–109.

[AHR15] Rob Alexander, Heather Rebecca Hawkins, and Andrew John Rae. *Situation coverage - a coverage criterion for testing autonomous robots*. Tech. rep. YCS-2015-496. Department of Computer Science, University of York, 2015.

[Alu15] Rajeev Alur. *Principles of Cyber-Physical Systems*. The MIT Press, 2015. ISBN: 9780262029117.

[Amo+16] Dario Amodei et al. “Concrete Problems in AI Safety”. In: *ArXiv* abs/1606.06565 (2016).

[Ana+13] Saswat Anand et al. “An orchestrated survey of methodologies for automated software test case generation”. In: *Journal of Systems and Software* 86.8 (2013), pp. 1978–2001.

[Ani+16] Adina Aniculaesei et al. “Towards the Verification of Safety-critical Autonomous Systems in Dynamic Environments”. In: *Proceedings of the The First Workshop on Verification and Validation of Cyber-Physical Systems, V2CPS@IFM 2016*. Ed. by Mehdi Kargahi and Ashutosh Trivedi. Vol. 232. EPTCS. 2016, pp. 79–90.

[Ani+18a] Adina Aniculaesei et al. “Automated Generation of Requirements-based Test Cases for an Adaptive Cruise Control System”. In: *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. Mar. 2018, pp. 11–15.

[Ani+18b] Adina Aniculaesei et al. “Towards a Holistic Software Systems Engineering Approach for Dependable Autonomous Systems”. In: *Proceedings of the 1st International Workshop on Software Engineering for AI in Autonomous Systems*. ACM, 2018, pp. 23–30.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. “Mining Sequential Patterns”. In: *Proceedings of the Eleventh International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 3–14.

[Avi+04] Algirdas Avivienis et al. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[Bac+15] Johannes Bach et al. “Control based driving assistant functions’ test using recorded in field data”. In: *Proc. 7. Tagung Fahrerassistenzsysteme*. Munich, Germany: TÜV SÜD Akad., 2015.

[Bac+17a] Johannes Bach et al. “Data-Driven Development, A Complementing Approach for Automotive Systems Engineering”. In: *2017 IEEE International Systems Engineering Symposium*. IEEE Computer Society, Oct. 2017, pp. 1–6.

[Bac+17b] Johannes Bach et al. “Reactive-Replay Approach for Verification and Validation of Closed-Loop Control Systems in Early Development”. In: *SAE Technical Paper Series*. 2017-01-1671. SAE International, Mar. 2017.

[Bac+17c] Johannes Bach et al. “Test Scenario Selection for System-Level Verification and Validation of Geolocation-Dependent Automotive Control Systems”. In: *2017 International Conference on Engineering, Technology and Innovation*. IEEE, June 2017, pp. 203–210.

[Ban+05] Jerry Banks et al. *Discrete-Event System Simulation*. Pearson Prentice-Hall, 2005. ISBN: 7111171942.

[Bar+09] Arne Bartels et al. “Qualitätsgesicherte Fahrentscheidungsunterstützung für automatisches Fahren auf Schnellstraßen und Autobahnen”. In: *Proceedings des 10. Braunschweiger Symposiums Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel*. Gesamtzentrum für Verkehr Braunschweig e.V., 2009, pp. 341–353.

[Bar+15] Earl T. Barr et al. “The Oracle Problem in Software Testing: A Survey”. In: *IEEE Transactions on Software Engineering* 41.5 (May 2015), pp. 507–525.

[Bar+16] Yvonne Barnard et al. “Methodology for Field Operational Tests of Automated Vehicles”. In: *Transportation Research Procedia* 14 (2016), pp. 2188–2196.

[Bar10] Jaume Barceló. “Models, Traffic Models, Simulation, and Traffic Simulation”. In: *Fundamentals of Traffic Simulation*. Ed. by Jaume Barceló. New York, NY: Springer New York, 2010, pp. 1–62. ISBN: 978-1-4419-6142-6.

[Bau+07] Thomas Bauer et al. “From Requirements to Statistical Testing of Embedded Systems”. In: *Fourth International Workshop on Software Engineering for Automotive Systems*. Washington, DC, USA: IEEE, May 2007, pp. 3–.

[Bau+08] Thomas Bauer et al. “Risikobasierte Ableitung und Priorisierung von Testfällen für den modellbasierten Systemtest”. In: *Software Engineering 2008. Fachtagung des GI-Fachbereichs Softwaretechnik*. Ed. by Korbinian Herrmann and Bernd Brügge. Vol. 121. LNI. Munich, Germany: GI, Feb. 2008, pp. 99–111.

[Bau+12] Eric Bauer et al. “PRORETA 3: An Integrated Approach to Collision Avoidance and Vehicle Automation”. In: *At-Automatisierungstechnik* 60.12 (Dec. 2012), pp. 755–765.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Ed. by M. Henzinger et al. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag Berlin Heidelberg, 2004.

[BC10] Yvonne Barnard and Oliver Carsten. “Field operational tests: challenges and methods”. In: *Proceedings of European conference on human centred design for intelligent transport systems*. Ed. by J. Krems, T. Petzholdt, and M. Henning. Lyon, France: HUMANIST Publications, 2010, pp. 323–332.

[BD08] Christopher R. Baker and John M. Dolan. “Traffic interaction in the urban challenge: Putting boss on its best behavior”. In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Nice, France: IEEE, Sept. 2008, pp. 1752–1758.

[Bec+14] Kristian Beckers et al. “Systematic Derivation of Functional Safety Requirements for Automotive Systems”. In: *Computer Safety, Reliability, and Security*. Ed. by A. Bondavalli and F. Di Giandomenico. Vol. 8666. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 65–80.

[Bei12] Sven A. Beiker. “Legal Aspects of Autonomous Driving”. In: *Santa Clara Law Review* 52.4 (2012), pp. 1145–1561.

[Bel+12] Assia Belbachir et al. “Simulation-Driven Validation of Advanced Driving-Assistance Systems”. In: *Transport Research Arena 2012*. Ed. by Panos Papaioannou. Vol. 48. Procedia-Social and Behavioral Sciences. Elsevier BV, 2012, pp. 1205–1214.

[Bel57] Richard Bellman. “A Markovian decision process”. In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684.

[Ben+14] Klaus Bengler et al. “Three Decades of Driver Assistance Systems: Review and Future Perspectives”. In: *IEEE Intelligent Transportation Systems Magazine* 6.4 (2014), pp. 6–22.

[Ber+14] Christian Berger et al. “Simulations on Consumer Tests: Systematic Evaluation of Tolerance Ranges by Model-Based Generation of Simulation Scenarios”. In: *Proc. Fahrerassistenzsysteme und Integrierte Sicherheit*. Vol. 2223. VDI Berichte 2014. VDI-Verlag, 2014, pp. 403–418.

[Ber+15] Christian Berger et al. “Simulations on Consumer Tests: A Systematic Evaluation Approach in an Industrial Case Study”. In: *IEEE Intelligent Transportation Systems Magazine* 7.4 (Oct. 2015), pp. 24–36. ISSN: 1939-1390.

[Ber10] Christian Berger. “Automating Acceptance Tests for Sensor-and Actuator-based Systems on the Example of Autonomous Vehicles”. In: *Aachener Informatik-Berichte, Software Engineering*. Ed. by Prof. Dr. rer. nat. Bernhard Rumpe. Vol. 6. Shaker Verlag, 2010.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software. The KeY Approach*. Springer Berlin Heidelberg, 2007. ISBN: 9783540689775.

[Bir+13] John Birch et al. “Safety cases and their role in ISO 26262 functional safety assessment”. In: *International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2013)*. Ed. by F. Bitsch, J. Guiochet, and M. Kaâniche. Vol. 8153. Lecture Notes in Computer Science. Springer. 2013, pp. 154–165.

[Bis96] Robert H. Bishop. *Modern Control Systems Analysis and Design Using MATLAB and SIMULINK*. First. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1996. ISBN: 0201498464.

[Bit01] Friedemann Bitsch. “Safety patterns - the key to formal specification of safety requirements”. In: *International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2001)*. Ed. by U. Voges. Vol. 2187. Lecture Notes in Computer Science. Springer, 2001, pp. 176–189.

[BJ05] Guillaume Brat and Ari Jonsson. “Challenges in verification and validation of autonomous systems for space exploration”. In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks*. Vol. 5. IEEE, 2005, pp. 2909–2914.

[BKM10] David Basin, Felix Klaedtke, and Samuel Müller. “Policy Monitoring in First-Order Temporal Logic”. In: *Computer Aided Verification*. Ed. by T. Touili, B. Cook, and P. Jackson. Vol. 6174. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, 2010, pp. 1–18.

[BKV13] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. “From propositional to first-order monitoring”. In: *International Conference on Runtime Verification (RV 2013)*. Ed. by A. Legay and S. Bensalem. Vol. 8174. Lecture Notes in Computer Science. Springer. Springer Berlin Heidelberg, 2013, pp. 59–75.

[BLS06] Andreas Bauer, Martin Leucker, and Christian Schallhart. “Model-based runtime analysis of distributed reactive systems”. In: *Australian Software Engineering Conference (ASWEC'06)*. IEEE, 2006, pp. 243–252.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. “Runtime Verification for LTL and TLTL”. In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (Sept. 2011), 14:1–14:64. ISSN: 1049-331X.

[BMF07] Thomas Bokc, Markus Maurer, and Georg Farber. “Validation of the Vehicle in the Loop (VIL); A milestone for the simulation of driver assistance systems”. In: *2007 IEEE Intelligent Vehicles Symposium*. IEEE, June 2007, pp. 612–617.

[BMW18] BMW AG. *Heading for Europe's motorways in a highly automated BMW. BMW Group and Continental team up on next step towards highly automated driving.* Online. https://www.press.bmwgroup.com/global/article/detail/T0137270EN/heading-for-europe-s-<-in-a-highly-automated-bmw-bmw-group-and-continental-team-up-on-next-step-towards-highly-automated-driving?language=en. Retrieved: 11/ 21/2018. 2018.

[BN96] Ruzena Bajcsy and Hans-Hellmut Nagel. “Descriptive and prescriptive languages for mobility tasks: Are they different”. In: *Advances in Image Understanding – A Festschrift for Azriel Rosenfeld*. Ed. by K. Bowyer and N. Ahuja. Los Alamitos, CA: IEEE Computer Society Press, 1996, pp. 280–300.

[BNF16] Guy Berg, Verena Nitsch, and Berthold Färber. “Vehicle in the Loop”. In: *Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort*. Ed. by Hermann Winner et al. Cham: Springer International Publishing, 2016, pp. 199–210. ISBN: 978-3-319-12352-3.

[BNG06] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. “Toward open-world software: Issues and challenges”. In: *Computer* 39.10 (Oct. 2006), pp. 36–43. ISSN: 00189162.

[Boc09] Thomas Bock. “Bewertung von Fahrerassistenzsystemen mittels der Vehicle in the Loop-Simulation”. In: *Handbuch Fahrerassistenzsysteme*. Ed. by H. Winner. Vieweg+Teubner, 2009, pp. 76–83.

[Böd+18] Eckard Böde et al. “Efficient Splitting of Test and Simulation Cases for the Verification of Highly Automated Driving Functions”. In: *Computer Safety, Reliability, and Security (SAFECOMP 2018)*. Ed. by Barbara Gallina, Amund Skavhaug, and Friedemann Bitsch. Vol. 11093. Lecture Notes in Computer Science, Springer, 2018, pp. 139–153. ISBN: 978-3-319-99130-6.

[Bon+96] R. Peter Bonasso et al. “Experiences with an architecture for intelligent, reactive agents”. In: *Intelligent Agents II Agent Theories, Architectures, and Languages (ATAL 1995)*. Ed. by M. Wooldridge, J. P. Müller, and M. Tambe. Vol. 1037. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 187–202. ISBN: 3540608052.

[BOS16] Johannes Bach, Stefan Otten, and Eric Sax. “Model based scenario specification for development and test of automated driving functions”. In: *2016 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, June 2016, pp. 1149–1155.

[BP09] Karsten Berns and Ewald von Puttkamer. *Autonomous Land Vehicles*. First. Vieweg+Teubner, 2009.

[BR15] Arne Bartels and Thomas Ruchatz. “Einführungsstrategie des Automatischen Fahrens”. In: *at - Automatisierungstechnik* 63.3 (Jan. 2015), pp. 168–179.

[Bra+15] Guillaume Brat et al. “Verifying the safety of a flight-critical system”. In: *International Symposium on Formal Methods (FM 2015)*. Ed. by N. Bjørner and F. de Boer. Vol. 9109. Lecture Notes in Computer Science, Springer. 2015, pp. 308–324.

[Bro03] Manfred Broy. “Automotive Software Engineering”. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE ’03)*. New York, New York, USA: IEEE Computer Society, 2003, pp. 719–720.

[Bro86] Rodney A. Brooks. “A robust layered control system for a mobile robot”. In: *IEEE Journal on Robotics and Automation* 2.1 (1986), pp. 14–23. ISSN: 0882-4967.

[Bru+09] Yuriy Brun et al. “Engineering self-adaptive systems through feedback loops”. In: *Software engineering for self-adaptive systems*. Ed. by B.H.C. Cheng et al. Vol. 5525. Lecture Notes in Computer Science. Springer, 2009, pp. 48–70.

[BT93] Dominique Briere and Pascal Traverse. “AIRBUS A320/A330/A340 electrical flight controls - A family of fault-tolerant systems”. In: *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE. IEEE Comput. Soc. Press, 1993, pp. 616–623.

[Bub02] Heiner Bubb. “Der Fahrprozess, Informationsverarbeitung durch den Fahrer”. In: *4. Technischer Kongress 20./21.* Stuttgart, Germany: VDA, 2002, pp. 19–38.

[Bud+96] Frank J. Budinsky et al. “Automatic code generation from design patterns”. In: *IBM Systems Journal* 35.2 (1996), pp. 151–171.

[BV10] Marco Bozzano and Adolfo Villafiorita. *Design and safety assessment of critical systems*. First. Auerbach Publications, 2010.

[Cap+13] Pantelis Capros et al. *EU Energy, Transport and GHG Emissions: Trends to 2050, reference scenario 2013*. Tech. rep. European Comimission: Directorate-General for Energy, Directorate-General for Climate Action, Directorate-General for Mobility, and Transport, 2013.

[Cas+87] Paul Caspi et al. “LUSTRE: a declarative language for real-time programming”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1987, pp. 178–188.

[CC04] David E. Clark and Brad M. Cushing. “Rural and urban traffic fatalities, vehicle miles, and population density”. In: *Accident Analysis & Prevention* 36.6 (Nov. 2004), pp. 967–972.

[CDS02] Szyperski Clemens, Gruntz Dominik, and Murer Stephan. *Component Software: Beyond Object-Oriented Programming*. Second Edition. Addison-Wesley Professional, 2002. ISBN: 0201745720.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. “Model checking and abstraction”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.5 (Sept. 1994), pp. 1512–1542.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. Ed. by Calin Belta. First. Vol. 2. Cyber Physical Systems Series. MIT press, 1999.

[CH03] Krzysztof Czarnecki and Simon Helsen. “Classification of model transformation approaches”. In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. USA. 2003.

[Cho05] Gobinda G. Chowdhury. “Natural language processing”. In: *Annual Review of Information Science and Technology* 37.1 (Jan. 2005), pp. 51–89.

[Chr02] Nicholas R. Chrisman. *Exploring Geographic Information System*. English. Wiley, 2002. ISBN: 0471314250.

[Chr08] Jost-Pieter Katoen Christel Baier. *Principles of Model Checking*. The MIT Press, June 11, 2008. ISBN: 9780262026499.

[Cim+02] Alessandro Cimatti et al. “NuSMV 2: An OpenSource Tool for Symbolic Model Checking”. In: *Computer Aided Verification (CAV 2002)*. Ed. by Ed Brinksma and Kim Guldstrand Larsen. Vol. 2404. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 359–364.

[Cim+08] Alessandro Cimatti et al. “From informal requirements to property-driven formal validation”. In: *Formal Methods for Industrial Critical Systems*. Ed. by D. Cofer and A. Fantechi. Vol. 5596. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 166–181.

[Cim+10] Alessandro Cimatti et al. “Formalization and Validation of Safety-Critical Requirements”. In: *16th Int. Symposium on Formal Methods*. Vol. 20. Open Publishing Association, Mar. 2010, pp. 68–75.

[Cla+99] Manuel Clavel et al. “The Maude System”. In: *10th International Conference on Rewriting Techniques and Applications*. Springer. 1999, pp. 240–243.

[Cle07] Jane Cleland-Huang. “Quality Requirements and their Role in Successful Products”. In: *15th IEEE International Requirements Engineering Conference (RE 2007)*. Ed. by Alistair Sutcliffe and Pankaj Jalote. IEEE. IEEE, Oct. 2007, pp. 361–361.

[Coh+97] Don Cohen et al. “Automatic Monitoring of Software Requirements”. In: *Proceedings of the 19th International Conference on Software Engineering*. New York, NY, USA: ACM, 1997, pp. 602–603.

[Con+06] James Connelly et al. “Current challenges in autonomous vehicle development”. In: *Unmanned Systems Technology VIII*. Ed. by Grant R. Gerhart, Charles M. Shoemaker, and Douglas W. Gage. Vol. 6230. SPIE, May 2006.

[CS217] C/S2ESC - Software & Systems Engineering Standards Committee. *IEEE 1012-2016 - IEEE Standard for System, Software, and Hardware Verification and Validation*. Tech. rep. C - IEEE Computer Society, 2017.

[CW96] Edmund M. Clarke and Jeannette M. Wing. “Formal Methods: State of the Art and Future Directions”. In: *ACM Computing Surveys (CSUR)* 28.4 (1996), pp. 626–643.

[DB10] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems*. 12th Edition. Pearson, 2010. ISBN: 9780136024583.

[Den+14] Louise A. Dennis et al. “Practical verification of decision-making in agent-based autonomous systems”. In: *Automated Software Engineering* 23.3 (Sept. 2014), pp. 305–359.

[DG15] Marko Dimjašević and Dimitra Giannakopoulou. “Test-case Generation for Runtime Analysis and Vice Versa: Verification of Aircraft Separation Assurance”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 282–292.

[DGH92] Paul Dagum, Adam Galper, and Eric Horvitz. “Dynamic Network Models for Forecasting”. In: *Proceedings of the Eighth International Conference on Uncertainty in Artificial Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, pp. 41–48.

[Dic07] Ernst Dieter Dickmanns. *Dynamic Vision for Perception and Control of Motion*. Springer London, 2007. ISBN: 978-1-84628-637-7.

[DMB04] Jie Du, James Masters, and Matthew J. Barth. "Lane-level positioning for in-vehicle navigation and automated vehicle location (AVL) systems". In: *7th International IEEE Conference on Intelligent Transportation Systems*. IEEE, Oct. 2004, pp. 35–40.

[Don+07] Erik Donner et al. "RESPONSE 3 - Code of Practice für die Entwicklung, Validierung und Markteinführung von Fahrerassistenzsystemen". In: *VDA Technischer Kongress*. 2007, pp. 231–241.

[Don15] Edmund Donges. "Driver Behavior Models". In: *Handbook of Driver Assistance Systems*. Ed. by Hermann Winner et al. Springer International Publishing, 2015, pp. 19–33.

[Don82] Edmund Donges. "Aspekte der aktiven Sicherheit bei der Führung von Personenkraftwagen". In: *Automobil-Industrie* 27.2 (1982), pp. 183–190.

[DT16] Nabarun Das and William Taylor. "Quantified Fault Tree Techniques for Calculating Hardware Fault Metrics According to ISO 26262". In: *2016 IEEE Symposium on Product Compliance Engineering*. IEEE, May 2016, pp. 1–8.

[Dup16] Marius Dupuis. *Openscenario - bringing content to the road*. Tech. rep. 2nd OpenSCENARIO Meeting, 2016.

[Dur05] Hugh Durrant-Whyte. "Autonomous land vehicles". In: *Proceedings of the Institution of Mechanical Engineers, Part 1: Journal of Systems and Control Engineering*. Vol. 219. 1. SAGE Publications, Feb. 2005, pp. 77–98.

[EGA13] EGAS Workgroup. *Standardized E-GAS Monitoring Concept for Gasoline and Diesel Engine Control Unit*. IAV GmbH, 2013.

[El 02] Mohammed El Shobaki. "On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems". In: *Proceedings of the 8th international conference on real-time computing systems and applications*. IEEE, 2002.

[ELC12] Heinz Erzberger, Todd A. Lauderdale, and Yung-Cheng Chu. "Automated conflict resolution, arrival management, and weather avoidance for air traffic management". In: *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering* 226.8 (Oct. 2012), pp. 930–949.

[ELS10] Christopher Edwards, Thomas Lombaerts, and Hafid Smaili, eds. *Fault Tolerant Flight Control*. Vol. 399. Lecture Notes in Control and Information Sciences. Springer, 2010.

[Eri05] Clifton A. Ericson. *Hazard Analysis Techniques for System Safety*. John Wiley & Sons, Inc., 2005. ISBN: 9780471720195.

[Eur11] European Commission. “Roadmap to a single European transport area - Towards a competitive and resource efficient transport system”. In: *White Paper (COM (2011) 144)* (2011).

[FDW13] Michael Fisher, Louise Dennis, and Matt Webster. “Verifying Autonomous Systems”. In: *Communications of the ACM* 56.9 (2013), pp. 84–93.

[FE98] Peter Fritzson and Vadim Engelson. “Modelica – A Unified Object-Oriented Language for System Modeling and Simulation”. In: *European Conference on Object-Oriented Programming*. Vol. 1445. Lecture Notes in Computer Science. Springer Berlin Heidelberg, July 1998, pp. 67–90.

[Fea+98] Martin S. Feather et al. “Reconciling System Requirements and Runtime Behavior”. In: *Proceedings Ninth International Workshop on Software Specification and Design*. IEEE Computer Society, Apr. 1998, pp. 50–59.

[Fed17] Federal Ministry of Transport and Digital Infrastructure. *Ethics Commission: Automated and connected driving*. June 2017.

[FG96] Stan Franklin and Art Graesser. “Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents”. In: *Intelligent Agents III. Agent Theories, Architectures, and Languages*. Ed. by Jörg Müller, Michael J. Wooldridge, and Nicholas Jennings. Vol. 1193. Lecture Notes in Artificial Intelligence. Springer-Verlag Berlin Heidelberg, 1996, pp. 21–35.

[FGG97] Nir Friedman, Dan Geiger, and Moises Goldszmidt. “Bayesian Network Classifiers”. In: *Machine Learning*. Vol. 29. 2-3. Kluwer Academic Publishers, Nov. 1997, pp. 131–163.

[FGT11] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. “A formal approach to adaptive software: continuous assurance of non-functional requirements”. In: *Formal Aspects of Computing*. Vol. 24. 2. Springer Nature, Nov. 2011, pp. 163–186.

[Fle05] FlexRay Consortium. *FlexRay Communications System Protocol Specification v2.1*. 2005.

[Flo67] Robert W. Floyd. “Nondeterministic Algorithms”. In: *Journal of the ACM*. Ed. by Victor Vianu. Vol. 14. 4. New York, NY, USA: ACM, Oct. 1967, pp. 636–644.

[For08] Forschungsgesellschaft für Straßen- und Verkehrswesen, Arbeitsgruppe Stra"senentwurf. “Richtlinien für die Anlage von Autobahnen: RAA”. In: *FGSV*. Vol. 220. Köln: FGSV-Verlag, 2008.

[FPE14] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. 7th Edition. Pearson, 2014. ISBN: 0133496597.

[Fra+17] Adrian Francalanza et al. “A Foundation for Runtime Monitoring”. In: *International Conference on Runtime Verification*. Ed. by S. Lahiri and G. Reger. Vol. 10548. Lecture Notes in Computer Science. Springer. 2017, pp. 8–29.

[Fra+18] Laura Fraade-Blanar et al. *Measuring Automated Vehicle Safety: Forging a Framework*. Tech. rep. RR-2662. Santa Monica: Calif.: RAND Corporation, Oct. 2018.

[Fri+06] Peter Fritzson et al. "OpenModelica – A Free Open-Source Environment for System Modeling, Simulation, and Teaching". In: *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design*. IEEE, Oct. 2006, pp. 1588–1595.

[Für+09] Simon Fürst et al. "AUTOSAR – A Worldwide Standard is on the Road". In: *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*. Vol. 2075. VDI Berichte. VDI, 2009, pp. 797–812.

[Gäf+08] Magnus Gäfvert et al. "Simulation-Based Automated Verification of Safety-Critical Chassis-Control Systems". In: *Proc. 9th Intern. Symposium on Advanced Vehicle Control*. Oct. 2008.

[Gas+12] Tom Michael Gasser et al. "Rechtsfolgen zunehmender Fahrzeugautomatisierung - Gemeinsamer Schlussbericht der Projektgruppe". In: *Berichte der Bundesanstalt für Straßenwesen*. Vol. 83. Wirtschaftsverlag NW, 2012.

[Gas12] Tom Michael Gasser. "Legal Issues of Driver Assistance Systems and Autonomous Driving". In: *Handbook of Intelligent Vehicles*. Ed. by Azim Eskandarian. London: Springer, 2012, pp. 1519–1535.

[GBH12] Philipp Glauner, Axel Blumenstock, and Martin Haueis. "Effiziente Felderprobung von Fahrerassistenzsystemen". In: *FAS, Workshop Fahrerassistenzsystem*. Vol. 8. Uni-DAS, Sept. 2012, pp. 5–14.

[Ger+14] Mario Gerla et al. "Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds". In: *2014 IEEE World Forum on Internet of Things (WF-IoT)*. IEEE. 2014, pp. 241–246.

[Gey+14] Sebastian Geyer et al. "Concept and development of a unified ontology for generating test and use-case catalogues for assisted and automated vehicle guidance". In: *IET Intelligent Transport Systems* 8.3 (2014), pp. 183–189.

[GG14] Heinrich Gotzig and Georg Geduld. "Automotive LIDAR". In: *Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort*. Ed. by Hermann Winner et al. Springer International Publishing, 2014, pp. 1–20.

[GGD12] Dominique Gruyer, Mélanie Grapinet, and Philippe De Souza. "Modeling and validation of a new generic virtual optical sensor for ADAS prototyping". In: *IEEE Intelligent Vehicles Symposium*. IEEE. 2012, pp. 969–974.

[Gha+18] Mohamad Gharib et al. "On the Safety of Automotive Systems Incorporating Machine Learning Based Components: A Position Paper". In: *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*. IEEE, 2018, pp. 271–274.

[Gia+14] Dimitra Giannakopoulou et al. “Taming Test Inputs for Separation Assurance”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 373–384.

[Gie+04] Olaf J. Gietelink et al. “Pre-crash System Validation with PRESCAN and VEHIL”. In: *IEEE Intelligent Vehicles Symposium* (2004), pp. 913–918.

[Gie+06] Olaf J. Gietelink et al. “Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations”. In: *Vehicle System Dynamics* 44.7 (2006), pp. 569–590.

[Goe10] Anita Goel. *Computer Fundamentals*. Pearson Education India, 2010. ISBN: 8131733092.

[GP10] Alwyn E Goodloe and Lee Pike. *Monitoring Distributed Real-Time Systems: A Survey and Future Directions*. Tech. rep. NASA Langley Reasearch Center, July 2010.

[GP17] Ian Goodfellow and Nicolas Papernot. *The challenge of verification and testing of machine learning*. Online. http://www.cleverhans.io/security/privacy/ml/2017/06/14/verification.html. Retrieved: 12/13/2018. 2017.

[Gri+01] Wolfgang Grieskamp et al. “Testable Use Cases in the Abstract State Machine Language”. In: *Proceedings Second Asia-Pacific Conference on Quality Software*. IEEE, 2001, pp. 167–172.

[GRS14] Clément Galko, Romain Rossi, and Xavier Savatier. “Vehicle-Hardware-In-The-Loop System for ADAS Prototyping and Validation”. In: *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. IEEE, 2014, pp. 329–334.

[Gru+10] Dominique Gruyer et al. “A new generic virtual platform for cameras modeling”. In: *Vehicle and Infrastructure Safety Improvement in Adverse Conditions and Night Driving* (Oct. 2010).

[Har87] David Harel. “Statecharts: A Visual Formalism For Complex Systems”. In: *Science of Computer Programming* 8.3 (1987), pp. 231–274.

[Hav11] Klaus Havelund. “Implementing Runtime Monitors”. In: *2nd TORRENTS Workshop* (2011).

[HB82] Steve J. Heims and Duane W. Bailey. “John von Neumann and Norbert Wiener, from Mathematics to the technologies of life and death”. In: *American Journal of Physics* 50.4 (1982), pp. 383–383.

[HBS18] Andreas Herrmann, Walter Brenner, and Rupert Stadler. *Autonomous Driving: How the Driverless Revolution Will Change the World*. Emerald Publishing Limited, 2018. ISBN: 1787148343.

[Hec+11] Karl Heckemann et al. “Safe Automotive Software”. In: *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*. Ed. by A. König et al. Vol. 6884. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2011, pp. 167–176.

[HG08] Klaus Havelund and Allen Goldberg. “Verify Your Runs”. In: *Verified Software: Theories, Tools, Experiments*. Ed. by B. Meyer and J. Woodcock. Vol. 4171. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Oct. 2008, pp. 374–383.

[HGR13] Falk Howar, Dimitra Giannakopoulou, and Zvonimir Rakamarić. “Hybrid Learning: Interface Generation through Static, Dynamic, and Symbolic Analysis”. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 268–279.

[Hil+11] Martin Hilscher et al. “An Abstract Model for Proving Safety of Multi-Lane Traffic Manoeuvres”. In: *Formal Methods and Software Engineering*. Ed. by Qin S. and Qiu Z. Vol. 6991. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 404–419.

[HK16] Stephan Hakuli and Markus Krug. “Virtual Integration in the Development Process of ADAS”. In: *Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort*. Ed. by Hermann Winner et al. Springer International Publishing, 2016, pp. 159–176.

[HLO13] Martin Hilscher, Sven Linker, and Ernst-Rüdiger Olderog. “Proving Safety of Traffic Manoeuvres on Country Roads”. In: *Theories of Programming and Formal Methods*. Ed. by Z. Liu, J. Woodcock, and H Zhu. Vol. 8051. Lecture Notes in Computer Science. Springer, 2013, pp. 196–212.

[HLR94] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. “Synchronous observers and the verification of reactive systems”. In: *Algebraic Methodology and Software Technology*. Ed. by Maurice Nivat et al. Workshops in Computing. Springer, London, 1994, pp. 83–96.

[HLZ17] Zhiyuan Huang, Henry Lam, and Ding Zhao. “Towards Affordable On-track Testing for Autonomous Vehicle - A Kriging-based Statistical Approach”. In: *IEEE 20th International Conference on Intelligent Transportation Systems*. IEEE, 2017, pp. 1–6.

[HMA03] Hui-Min Huang, Elena Messina, and James Albus. “Autonomy Level Specification for Intelligent Autonomous Vehicles: Interim Progress Report”. In: *2003 Performance Metrics for Intelligent Systems (PerMIS)* September (2003), pp. 1–7.

[HMF14] Donal Heffernan, Ciaran MacNamee, and Padraig Fogarty. “Runtime verification monitoring for automotive embedded systems using the ISO 26262 Functional Safety Standard as a guide for the definition of the monitored properties”. In: *IET Software* 8.5 (Oct. 2014), pp. 193–203.

[Hör11] Markus Hörwick. “Sicherheitskonzept für hochautomatisierte Fahrerassistenzsysteme”. PhD thesis. Technische Universität München, 2011.

[HPT10] Falke Hendriks, Riné Pelders, and Martijn Tideman. “Future Testing of Active Safety Systems”. In: *SAE International Journal of Passenger Cars - Electronic and Electrical Systems* 3.2 (Oct. 2010), pp. 170–175.

[HR04] Michael Huth and Mark Ryan. “Predicate logic”. In: *Logic in Computer Science: Modelling and Reasoning about Systems*. 2nd Edition. Cambridge University Press, 2004, pp. 93–171. ISBN: 052154310X.

[HRS98] Kirsten M. Hansen, Anders P. Ravn, and Victoria Stavridou. “From Safety Analysis to Software Requirements”. In: *IEEE Transactions on Software Engineering*. Vol. 24. 7. IEEE, July 1998, pp. 573–584.

[HW08] Mordechai Haklay and Patrick Weber. “OpenStreetMap: User-Generated Street Maps”. In: *IEEE Pervasive Computing*. Vol. 7. 4. IEEE Computer Society, Oct. 2008, pp. 12–18.

[HW11] Kai Homeier and Lars Wolf. “RoadGraph: High level sensor data fusion between objects and street network”. In: *14th International IEEE Conference on Intelligent Transportation Systems*. IEEE, Oct. 2011, pp. 1380–1385.

[HZB00] Almut Hochstädter, Peter Zahn, and Karsten Breuer. “Ein universelles Fahrermodell mit den Einsatzbeispielen Verkehrssimulation und Fahrsimulator”. In: *9. Aachener Kolloquium Fahrzeug- und Motorentechnik*. TH Aachen, Lehrstuhl für Verbrennungskraftmaschinen, Oct. 2000.

[Ina06] Toshiyuki Inagaki. “Design of human – machine interactions in light of domain-dependence of human-centered automation”. In: *Cognition, Technology & Work* 8.3 (Apr. 2006), pp. 161–167.

[Int09a] International Organization for Standardization. *Road vehicles - Functional safety - Part 1: Vocabulary*. 2009.

[Int09b] International Organization for Standardization. *Road vehicles - Functional safety - Part 10: Guideline*. 2009.

[Int09c] International Organization for Standardization. *Road vehicles - Functional safety - Part 3: Concept phase*. 2009.

[Int09d] International Organization for Standardization. *Road vehicles - Functional safety - Part 4: Product development: system level*. 2009.

[Int09e] International Organization for Standardization. *Road vehicles - Functional safety - Part 5: Product development: hardware level*. 2009.

[Int09f] International Organization for Standardization. *Road vehicles - Functional safety - Part 6: Product development: software level*. 2009.

[Int11a] International Organization for Standardization. *ISO 8855:2011: Road vehicles - Vehicle dynamics and road-holding ability - Vocabulary*. 2011.

[Int11b] International Organization for Standardization. *ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*. 2011.

[Int17] International Organization for Standardization. *ISO/IEC/IEEE International Standard - Systems and software engineering - Vocabulary*. Aug. 2017.

[Ise05] Rolf Isermann. "Model-based fault-detection and diagnosis - status and applications". In: *Annual Reviews in control*. Ed. by Janos J. Gertler et al. Vol. 29. 1. Elsevier BV, 2005, pp. 71–85.

[Ise06a] Rolf Isermann, ed. *Fahrdynamik-Regelung*. Vieweg+Teubner Verlag, 2006. ISBN: 978-3-8348-9049-8.

[Ise06b] Rolf Isermann. *Fault-Diagnosis Systems*. Springer Berlin Heidelberg, 2006. ISBN: 978-3-540-30368-8.

[Ise11] Rolf Isermann. *Fault-Diagnosis Applications*. Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-12766-3.

[Ise97] Rolf Isermann. "Supervision, Fault-Detection and Fault-Diagnosis Methods - An Introduction". In: *Control Engineering Practice*. Vol. 5. 5. Elsevier, 1997, pp. 639–652.

[Jak+15] Stefan Jakšić et al. "From signal temporal logic to FPGA monitors". In: *ACM/IEEE International Conference on Formal Methods and Models for Codesign*. IEEE, 2015, pp. 218–227.

[JBS13] Ethan K. Jackson, Nikolaj Bjorner, and Wolfram Schulte. *Open-World Logic Programs: A New Foundation for Formal Specifications*. Tech. rep. 2013-55. Microsoft Research, 2013.

[Kal17] Nidhi Kalra. *Challenges and Approaches to Realizing Autonomous Vehicle Safety*. Online. https://www.rand.org/pubs/testimonies/CT463.html. 2017.

[Kan+15] Aaron Kane et al. "A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System". In: *Runtime Verification*. Ed. by Ezio Bartocci and Rupak Majumdar. Vol. 9333. Lecture Notes in Computer Science. Springer. 2015, pp. 102–117.

[Kan15] Aaron Kane. "Runtime Monitoring for Safety-Critical Embedded Systems". PhD thesis. Carnegie Mellon University, 2015.

[Kap+16] James Kapinski et al. "Simulation-Based Approaches for Verification of Embedded Control Systems: An Overview of Traditional and Advanced Modeling, Testing, and Verification Techniques". In: *IEEE Control Systems* 36.6 (2016), pp. 45–64.

[KFK14] Aaron Kane, Thomas Fuhrman, and Philip Koopman. "Monitor Based Oracles for Cyber-Physical System Testing: Practical Experience Report". In: *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, June 2014, pp. 148–155.

[KH10] Jorn Knaup and Kai Homeier. "RoadGraph - Graph based environmental modelling and function independent situation analysis for driver assistance systems". In: *13th International IEEE Conference on Intelligent Transportation Systems*. IEEE, 2010, pp. 428–432.

[Kha+17] Siddartha Khastgir et al. “Test Scenario Generation for Driving Simulators Using Constrained Randomization Technique”. In: *SAE Technical Paper*. Vol. 2017-01-1672, SAE International, 2017.

[Kim+08] Seung-Han Kim et al. “A Gateway System for an Automotive System: LIN, CAN, and FlexRay”. In: *6th IEEE International Conference on Industrial Informatics*. IEEE. 2008, pp. 967–972.

[Kis02] Ivan Kiselev. *Aspect-Oriented Programming with AspectJ*. Indianapolis, IN, USA: Sams, 2002. ISBN: 0672324105.

[KK15] Fabian Kneer and Erik Kamsties. “Model-based Generation of a Requirements Monitor.” In: *Joint Proceedings of REFSQ-2015 Workshops, Research Method Track, and Poster Track co-located with the 21st International Conference on Requirements Engineering: Foundation for Software Quality*. Vol. 1342. CEUR Workshop Proceedings. 2015, pp. 156–170.

[KKL13] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. *Introduction to Combinatorial Testing*. Chapman and Hall/CRC, 2013. ISBN: 9781466552296.

[KLC98] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. “Planning and acting in partially observable stochastic domains”. In: *Artificial intelligence*. Ed. by R. Dechter and P. Doherty. Vol. 101. 1-2. Elsevier, May 1998, pp. 99–134.

[KM08] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. ISBN: 9783540674016.

[KMM07] Ingolf H. Krüger, Michael Meisinger, and Massimiliano Menarini. “Runtime Verification of Interactions: From MSCs to Aspects”. In: *Runtime Verification*. Ed. by Oleg Sokolsky and Serdar Taşiran. Vol. 4839. Lecture Notes in Computer Science. Springer. Springer Berlin Heidelberg, 2007, pp. 63–74.

[Kna+17a] Alessia Knauss et al. “Paving the Roadway for Safety of Automated Vehicles: An Empirical Study on Testing Challenges”. In: *IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2017, pp. 1873–1880.

[Kna+17b] Alessia Knauss et al. “Software-Related Challenges of Testing Automated Vehicles”. In: *Proceedings of the 39th International Conference on Software Engineering Companion*. [IEEE], 2017, pp. 328–330.

[Koy92] Ron Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Vol. 651. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 1992.

[KP16] Nidhi Kalra and Susan M. Paddock. *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* Online. https://www.rand.org/pubs/research_reports/RR1478.html. 2016, pp. 182–193.

[KW16] Philip Koopman and Michael Wagner. “Challenges in Autonomous Vehicle Testing and Validation”. In: *SAE International Journal of Transportation Safety* 4.1 (2016), pp. 2016–01–0128. ISSN: 2327-5634.

[Las+10] Yvonne Laschinsky et al. “Evaluation of an Active Safety Light using Virtual Test Drive within Vehicle in the Loop”. In: *IEEE International Conference on Industrial Technology* (2010).

[Lee+85] Woon Sik Lee et al. “Fault Tree Analysis, Methods, and Applications: A Review”. In: *IEEE Transactions on Reliability* R-34.3 (Aug. 1985), pp. 194–203.

[Lee82] Chun Sing George Lee. “Robot Arm Kinematics, Dynamics, and Control”. In: *Computer*. Vol. 15. 12. IEEE, Dec. 1982, pp. 62–80.

[Lef+97] Ulrich Lefarth et al. “ASCET-SD – Development Environment for Embedded Control Systems”. In: *IFAC Proceedings Volumes*. Vol. 30. 4. Elsevier BV, 1997, pp. 85–90.

[Leo+08] John Leonard et al. “A Perception-Driven Autonomous Urban Vehicle”. In: *Journal of Field Robotics* 25.10 (Oct. 2008), pp. 727–774.

[Leu11] Martin Leucker. “Teaching runtime verification”. In: *Runtime Verification*. Ed. by Sarfraz Khurshid and Koushik Sen. Vol. 7186. Lecture Notes in Computer Science. Springer. 2011, pp. 34–48.

[LF96] Kangsun Lee and Paul A. Fishwick. “Dynamic Model Abstraction”. In: *Proceedings of the 28th conference on Winter simulation*. Ed. by John M. Charnes et al. IEEE, 1996, pp. 764–771.

[LF97] Kangsun Lee and Paul A Fishwick. “Semiautomated method for dynamic model abstraction”. In: *Enabling Technology for Simulation Science*. Ed. by Alex F. Sisti. Vol. 3083. Proc. SPIE. SPIE, June 1997, pp. 31–42.

[LH94] Jaynarayan H Lala and Richard E Harper. “Architectural Principles for Safety-Critical Real-Time Applications”. In: *Proceedings of the IEEE*. Vol. 82. 1. IEEE, 1994, pp. 25–40.

[Lig09] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Springer Science & Business Media, 2009.

[LK97] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. 2nd. McGraw-Hill Higher Education, 1997. ISBN: 0070366985.

[LMT15] Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. “Safe and Optimal Adaptive Cruise Control”. In: *Correct System Design*. Ed. by Roland Meyer, André Platzer, and Heike Wehrheim. Vol. 9360. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 260–277. ISBN: 978-3-319-23506-6.

[LP10] Daniel Le Berre and Anne Parrain. “The sat4j library, release 2.2, system description”. In: *Journal on Satisfiability, Boolean Modeling and Computation*. Vol. 7. IOS Press, 2010, pp. 59–64.

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. “UPPAAL in a nutshell”. In: *International Journal on Software Tools for Technology Transfer*. Vol. 1. 1-2. Springer, 1997, pp. 134–152.

[LS09] Martin Leucker and Christian Schallhart. “A brief account of runtime verification”. In: *The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software*. Ed. by Olaf Owe and Gerardo Schneider. Vol. 78. The Journal of Logic and Algebraic Programming 5. Elsevier BV, 2009, pp. 293–303.

[LSK13] Ulrich Lages, Martin Spencer, and Roman Katz. “Automatic Scenario Generation based on Laserscanner Reference Data and Advanced Offline Processing”. In: *IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2013, pp. 153–155.

[Luc+16] Alberto Lucchetti et al. “Automatic recognition of driving scenarios for ADAS design”. In: *8th IFAC Symposium on Advances in Automotive Control*. Ed. by Per Tunestål and Lars Eriksson. Vol. 49. IFAC-PapersOnLine 11. 8th IFAC Symposium on Advances in Automotive Control AAC 2016. 2016, pp. 109–114.

[Mac+13] Frank D. Macías-Escrivá et al. “Self-adaptive systems: A survey of current approaches, research challenges and applications”. In: *Expert Systems with Applications* 40.18 (2013), pp. 7267–7279. ISSN: 0957-4174.

[Mac+14] Mathilde Machin et al. “Specifying Safety Monitors for Autonomous Systems Using Model-Checking”. In: *Computer Safety, Reliability, and Security*. Ed. by Andrea Bondavalli and Felicita Di Giandomenico. Vol. 8666. Lecture Notes in Computer Science. Springer. 2014, pp. 262–277.

[Mac+18] Mathilde Machin et al. “SMOF - A Safety MOnitoring Framework for Autonomous Systems”. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems*. Vol. 48. 5. IEEE, 2018, pp. 702–715.

[Mar+13] Riccardo Marino et al. “Fault-tolerant cruise control of electric vehicles with induction motors”. In: *Control Engineering Practice* 21.6 (2013), pp. 860–869.

[Mar10] Rebecca J. Parsons Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010. ISBN: 0321712943.

[Mar18] Gary Marcus. “Deep Learning: A Critical Appraisal”. In: *CoRR* abs/1801.00631 (2018).

[Mat15] Richard Matthaei. “Wahrnehmungsgestützte Lokalisierung in fahrstreifengenauen Karten für Assistenzsysteme und automatisches Fahren in urbaner Umgebung”. PhD thesis. Technische Universität Braunschweig, 2015.

[Mau00a] Markus Maurer. “EMS-Vision: Knowledge Represent at ion for Flexible Automation of Land Vehicles”. In: *Proceedings of the IEEE Intelligent Vehicles Symposium 2000*. IEEE, Oct. 2000, pp. 575–580.

[Mau00b] Markus Maurer. “Flexible Automatisierung von Straßenfahrzeugen mit Rechnersehen”. In: *Verkehrstechnik/Fahrzeugtechnik*. Vol. 12. 445. https://www.ifr.ing.tu-bs.de/static/files/forschung/buecher/dissertation_maurer.pdf. VDI-Verlag, 2000.

[ME10] Nizar R. Mabroukeh and C. I. Ezeife. “A Taxonomy of Sequential Pattern Mining Algorithms”. In: *ACM Comput. Surveys*. Vol. 43. 1. ACM, Dec. 2010, 3:1–3:41.

[Mer18] Merriam-Webster Online Dictionary. *Autonomy*. Online. https://www.merriam-webster.com/dictionary/autonomy. Retrieved: 11-8-2018. 2018.

[MFG11] Javad Mohammadpour, Matthew Franchek, and Karolos Grigoriadis. “A Survey on Diagnostics Methods for Automotive Engines”. In: *Int. Journal of Engine Research*. Vol. 13. 1. SAGE Publications, Nov. 2011, pp. 41–64.

[MG14] Shabnam Mousavi and Gerd Gigerenzer. “Risk, uncertainty, and heuristics”. In: *Journal of Business Research* 67.8 (2014), pp. 1671–1678. ISSN: 0148-2963.

[MHR15] Malte Mauritz, Falk Howar, and Andreas Rausch. “From Simulation to Operation : Using Design Time Artifacts to Ensure the Safety of Advanced Driving Assistance Systems at Runtime”. In: *International Workshop on Modelling in Automotive Software Engineering* (2015).

[MHR16] Malte Mauritz, Falk Howar, and Andreas Rausch. “Assuring the safety of advanced driver assistance systems through a combination of simulation and runtime monitoring”. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2016, pp. 672–687.

[Mon+08] Michael Montemerlo et al. “Junior: The Stanford Entry in the Urban Challenge”. In: *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*. Ed. by Martin Buehler, Karl Iagnemma, and Sanjiv Singh. Vol. 56. Springer Tracts in Advanced Robotics. Springer, Berlin, Heidelberg, 2008, pp. 91–123.

[Mon+97] Robert T Monroe et al. “Architectural Styles, Design Patterns, and Objects”. In: *IEEE Software*. Vol. 14. 1. IEEE, 1997, pp. 43–52.

[Moo97] Christopher Z. Mooney. *Monte carlo simulation*. Vol. 116. Quantitative Applications in the Social Sciences. Sage Publications, 1997. ISBN: 0803959435.

[MP+07] Charlie Miller, Zachary N. J. Peterson, et al. “Analysis of mutation and generation-based fuzzing”. In: *Independent Security Evaluators, Tech. Rep* (2007).

[MPB09] Daniel Meyer-Delius, Christian Plagemann, and Wolfram Burgard. “Probabilistic situation recognition for vehicular traffic scenarios”. In: *IEEE International Conference on Robotics and Automation 2009 (ICRA’09)*. Citeseer. 2009, pp. 459–464.

[MR82] Michael F. Morris and Paul F. Roth. *Computer Performance Evaluation: Tools and Techniques for Effective Analysis*. Van Nostrand Reinhold Company, 1982.

[MRS14] Malte Mauritz, Andreas Rausch, and Ina Schaefer. "Dependable ADAS by Combining Design Time Testing and Runtime Monitoring". In: *FORMS/ FORMAT 2014, 10th Int. Symp. on Formal Methods*. 2014, pp. 28–37.

[MS11] Frank Moosmann and Christoph Stiller. "Velodyne SLAM". In: *IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2011, pp. 393–398.

[MSB11] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[Nat98] National Research Council. *Statistics, Testing, and Defense Acquisition: New Approaches and Methodological Improvements*. Ed. by Micheal L. Cohen, Hohn E. Rolph, and Duane L. Steffey. Washington, DC: The National Academies Press, May 1998. ISBN: 978-0-309-06551-1.

[NDW09] Kilian von Neumann-Cosel, Marius Dupuis, and Christian Weiss. "Virtual test drive–provision of a consistent tool-set for [d, h, s, v]-in-the-loop". In: *Proceedings of the Driving Simulation Conference Monaco*. 2009.

[NE91] Hans-Hellmut Nagel and Wilfried Enkelmann. "Generic road traffic situations and driver support systems". In: *Proceedings of the 5th Prometheus Workshop*. Prometheus Office, 1991, pp. 76–85.

[Nen+15] Laura Nenzi et al. "Qualitative and quantitative monitoring of spatio-temporal properties". In: *Runtime Verification*. Springer. 2015, pp. 21–37.

[Neu14] Kilian von Neumann-Cosel. "Virtual Test Drive: Simulation umfeldbasierter Fahrzeugfunktionen". PhD thesis. Universitätsbibliothek der TU München, 2014.

[Ngu+16] Thang Nguyen et al. "The HARMONIA project: hardware monitoring for automotive systems-of-systems". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2016, pp. 371–379.

[NL14] Richard Ni and Jason Leung. *Safety and Liability of Autonomous Vehicle Technologies*. Tech. rep. Massachusetts Institute, 2014.

[NM08] Bernd Neumann and Ralf Möller. "On scene interpretation with description logics". In: *Image and Vision Computing* 26.1 (2008), pp. 82–101.

[Not14] Tobias Nothdurft. *Ein Kontextmodell für sicherheitsrelevante Anwendungen in der autonomen Fahrzeugführung*. Techn. Univ., Niedersächsisches Forschungszentrum für Luftfahrt, 2014.

[NPP15] Geoffrey Nelissen, David Pereira, and Luis Miguel Pinho. "A novel runtime monitoring architecture for safe and efficient inline monitoring". In: *Ada-Europe International Conference on Reliable Software Technologies*. Springer. 2015, pp. 66–82.

[NYC15] Anh Nguyen, Jason Yosinski, and Jeff Clune. “Deep neural networks are easily fooled: High confidence predictions for unrecognizable images”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 427–436.

[OB12] Bernd Oestereich and Stefan Bremer. *Analyse und Design mit der UML 2.5: objektorientierte Softwareentwicklung*. Oldenbourg verlag, 2012.

[Oli+16] Stephanie Prialé Olivares et al. “Virtual Stochastic Testing of Advanced Driver Assistance Systems”. In: *Advanced Microsystems for Automotive Applications 2015*. Springer, 2016, pp. 25–35.

[Ors+02] Alessandro Orso et al. “Gamma System: Continuous Evolution of Software after Deployment”. In: *ACM SIGSOFT Software Engineering Notes* 27.4 (2002), p. 65. ISSN: 01635948.

[Par62] Emanuel Parzen. “On estimation of a probability density function and mode”. In: *The annals of mathematical statistics* 33.3 (1962), pp. 1065–1076.

[Pat+17] Sujeet Milind Patole et al. “Automotive radars: A review of signal processing techniques”. In: *IEEE Signal Processing Magazine* 34.2 (2017), pp. 22–35.

[PD02] M Pellkofer and ED Dickmanns. “Behavior decision in autonomous vehicles”. In: *Intelligent Vehicle Symposium*. Vol. 2. IEEE. 2002, pp. 495–500.

[PDE11] Rahul Purandare, Matthew B Dwyer, and Sebastian Elbaum. “Monitoring finite state properties: Algorithmic approaches and their relative strengths”. In: *International Conference on Runtime Verification*. Springer. 2011, pp. 381–395.

[PGV12] Steve Pechberti, Dominique Gruyer, and Vincent Vigneron. “Radar simulation in SiVIC platform for transportation issues. Antenna and propagation channel modelling”. In: *15th International IEEE Conference on Intelligent Transportation Systems*. IEEE. 2012, pp. 469–474.

[PHK17] Günther Prokop, Larse Hannawald, and Markus Köbe. “Eine Bewertungsmethodik zur Inspektion automatisierter Fahrfunktionen. Szenarien-basierte Plattform zur Inspektion automatisierter Fahrfunktionen. Das Projekt SePIA”. In: *Hochschule fuer angewandte Wissenschaften Kempten, Schriftenreihe* 3 (2017).

[PHR16] Henrik Peters, Falk Howar, and Andreas Rausch. “Towards inferring environment models for control functions from recorded signal data”. In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 2. IEEE. 2016, pp. 1–4.

[PM06] Bureau International des Poids and Mesures. *International System of Units (SI)*. Sèvres, France: Le Bureau, 2006.

[Pnu77] Amir Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science*. IEEE. 1977, pp. 46–57.

[QON07] Mohammed A. Quddus, Washington Y. Ochieng, and Robert B. Noland. "Current map-matching algorithms for transport applications: State-of-the art and future research directions". In: *Transportation research part C: Emerging technologies* 15.5 (2007), pp. 312–328.

[Rad17] Sabine Radke, ed. *Verkehr in Zahlen 2017/2018*. Vol. 46. Bundesministerium für Verkehr und digitale Infrastruktur, 2017.

[Ray+09] Arnab Ray et al. "Validating automotive control software using instrumentation based verification". In: *24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*. IEEE. 2009, pp. 15–25.

[RB08] Andreas Rausch and Manfred Broy. "Die V-Modell XT Grundlagen". In: *Das V-Modell XT: Grundlagen, Methodik und Anwendungen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–27. ISBN: 978-3-540-30250-6.

[Rd04] Sérgio R. Rota and Jorge R. de Almeida. "Run-time monitoring for dependable systems: an approach and a case study". In: *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE. 2004, pp. 41–49.

[Rei+10] Michael Reichel et al. "Situation aspect modelling and classification using the scenario based random forest algorithm for convoy merging situations". In: *13th International IEEE Conference on Intelligent Transportation Systems*. IEEE. 2010, pp. 360–366.

[Rei+11] Thomas Reinbacher et al. "Past time LTL runtime verification for microcontroller binary code". In: *International Workshop on Formal Methods for Industrial Critical Systems*. Springer. 2011, pp. 37–51.

[RG05] Ralph H. Rasshofer and Klaus Gresser. "Automotive radar and lidar systems for next generation driver assistance functions". In: *Advances in Radio Science* 3.B. 4 (2005), pp. 205–209.

[Ric16] Thomas Richter. *Planung von Autobahnen und Landstraßen*. Springer, 2016.

[RM15] Andreas Reschka and Markus Maurer. "Conditions for a safe state of automated road vehicles". In: *it-Information Technology* 57.4 (2015), pp. 215–222.

[Rot+11] Erwin Roth et al. "Analysis and Validation of Perception Sensor Models in an Integrated Vehicle and Environment Simulation". In: *Proceedings of the 22nd Enhanced Safety of Vehicles Conference*. 2011.

[RP92] Colette Rolland and Christophe Proix. "A Natural Language Approach for Requirements Engineering". In: *g* 593 (1992), pp. 257–277.

[SAE14] SAE On-Road Automated Vehicle Standards Committee. "J3016-2014: Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems". In: *SAE Standard J* 3016 (2014), pp. 1–16.

[SAE18] SAE On-Road Automated Vehicle Standards Committee. “J3016-2018: Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems”. In: *SAE Standard J* 3016 (2018), pp. 1–16.

[Sau+09] Falko Saust et al. “Entwicklungsbegleitendes Simulations-und Testkonzept für autonome Fahrzeuge in städtischen Umgebungen”. In: *AAET 2009. Automatisierungs-, Assistenzsysteme und eingebettete Systeme für Transportmittel* 129 (2009).

[Sch+11] Hans-Peter Schöner et al. “Koordiniertes automatisiertes Fahren für die Erprobung von Assistenzsystemen”. In: *ATZ-Automobiltechnische Zeitschrift* 113.1 (2011), pp. 40–45.

[Sch+13] Fabian Schuldt et al. “Effiziente systematische Testgenerierung für Fahrerassistenzsysteme in virtuellen Umgebungen”. In: *AAET2013 - Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel,* (2013).

[Sch+14] Fabian Schuldt et al. “Systematische Auswertung von Testfällen für Fahrfunktionen im modularen virtuellen Testbaukasten”. In: *9. Workshop Fahrerassistenzsysteme*. 2014, pp. 169–179.

[Sch+17] Alexander Schaermann et al. “Validation of vehicle environment sensor models”. In: *IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2017, pp. 405–411.

[Sch05] Stephen Schmitt. *Integrierte Simulation und Emulation eingebetteter Hardware/Software-Systeme*. Cuvillier Verlag, 2005.

[Sch07] Roland Schabenberger. “ADTF: Framework for driver assistance and safety systems”. In: *VDI Berichte* 2000 (2007), p. 701.

[Sch10] Brent Schwarz. “LIDAR: Mapping the world in 3D”. In: *Nature Photonics* 4.7 (2010), p. 429.

[Sch11a] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[Sch11b] Frank Schroven. “Probabilistische Situationsanalyse für eine adaptive automatisierte Fahrzeuglängsführung”. PhD thesis. Technische Universität Braunschweig, 2011.

[Sch12] Florian Schmidt. “Funktionale Absicherung kamerabasierter Aktiver Fahrerassistenzsysteme durch Hardware-in-the-Loop-Tests”. PhD thesis. TU Kaiserslautern, 2012.

[Sch13] Wladimir Schamai. “Model-based verification of dynamic system behavior against requirements: Method, language, and tool”. PhD thesis. Linköping University, 2013.

[Sch17] Fabian Schuldt. “Ein Beitrag für den methodischen Test von automatisierten Fahrfunktionen mit Hilfe von virtuellen Umgebungen”. PhD thesis. Technische Universität Braunschweig, Apr. 2017.

[Sch95] Beth A. Schroeder. “On-Line Monitoring: A Tutorial”. In: *Computer* 28 (June 1995), pp. 72–78. ISSN: 0018-9162.

[Sha98] Robert E Shannon. “Introduction to the art and science of simulation”. In: *Proceedings of the 30th conference on Winter simulation*. IEEE Computer Society Press. 1998, pp. 7–14.

[Sie+11] Sebastian Siegl et al. “Automated testing of embedded automotive systems from requirement specification models”. In: *12th Latin American Test Workshop (LATW)*. IEEE. 2011, pp. 1–6.

[Sip+16] Christoph Sippl et al. “From simulation data to test cases for fully automated driving and ADAS”. In: *IFIP International Conference on Testing Software and Systems*. Springer. 2016, pp. 191–206.

[SJ12] Jingjing Shen and Xiaogang Jin. “Detailed traffic animation for urban road networks”. In: *Graphical Models* 74.5 (2012), pp. 265–282.

[Slo08] JJ Slob. “State-of-the-art driving simulators, a literature survey”. In: *DCT report* 107 (2008).

[SLS18] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. “Systematic mapping study of template-based code generation”. In: *Computer Languages, Systems & Structures* 52 (2018), pp. 43–62.

[Smu12] Raymond R Smullyan. *First-order logic*. Vol. 43. Springer Science & Business Media, 2012.

[Som+13] Stephan Sommer et al. “Race: A centralized platform computer based architecture for automotive applications”. In: *IEEE International Electric Vehicle Conference (IEVC)*. IEEE. 2013, pp. 1–6.

[SSS17] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. “On a formal model of safe and scalable self-driving cars”. In: *ArXiv* abs/1708.06374 (2017).

[SSW13] Mohsen Sefati, Alexander Stoff, and Hermann Winner. “Testing Method for Autonomous Safety Functions Based on Combined Steering/Braking Maneuvers for Collision Avoidance and Mitigation.” In: *6. Tagung Fahrerassistenz*. 2013.

[Ste+15] Jan Erik Stellet et al. “Testing of Advanced Driver Assistance Towards Automated Driving: A Survey and Taxonomy on Existing Approaches and Open Questions.” In: *ITSC*. 2015, pp. 1455–1462.

[Ste10] Angus Stevenson. “Autonomy”. In: *Oxford Dictionary of English*. Oxford University Press, Jan. 2010. ISBN: 9780199571123.

[Ste14] Milan Stevanovic. *Advanced C and C++ Compiling*. 1st. Berkely, CA, USA: Apress, 2014. ISBN: 9781430266679.

[Sti13] Matthias Stiller. “Absicherung von Systemen für das (hoch) automatisierte Fahren”. In: *6. Fachkonferenz Autotest*. 6. needs rework. Oct. 2013.

[Str12] Benedikt Strasser. *Vernetzung von Test-und Simulationsmethoden für die Entwicklung von Fahrerassistenzsystemen*. Cuvillier, E, 2012.

[Str16] Umberto Straccia. *Foundations of fuzzy logic and semantic web languages*. Chapman and Hall/CRC, 2016.

[Sun+15] Jun Sun et al. "TLV: abstraction through testing, learning, and validation". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 698–709.

[SV06] Johann Schumann and Willem Visser. "Autonomy Software: V&V Challenges and Characteristics". In: *2006 IEEE Aerospace Conference*. 2006, pp. 1–6.

[SZ13] Jörg Schäuffele and Thomas Zurawka. *Automotive software engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. Springer-Verlag, 2013.

[Szl18] Andreas Szlatki. "Test Case Generation for System Simulations of Advanced Driver Assistance Systems by Analysis of Recorded Field Data". MA thesis. University of Mannheim, Feb. 2018.

[Tat15] Mugur Tatar. "Enhancing ADAS test and validation with automated search for critical situations". In: *Driving Simulation Conference (DSC)*. 2015.

[Tel12] Dirk Tellmann. *Hardware-in-the-Loop-gestützte Entwicklungsplattform für Fahrerassistenzsysteme: Modelle der Umfeldsensorik und angepasste Fahrermodelle*. kassel university press GmbH, 2012.

[TH02] Dave Thomas and Andy Hunt. "Mock objects". In: *IEEE Software* 19.3 (May 2002), pp. 22–24. ISSN: 0740-7459.

[Tha+03] Henrik Thane et al. "Replay debugging of real-time systems using time machines". In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE. 2003, 8–pp.

[THH06] Sri Fatimah Tjong, Nasreddine Hallam, and Michael Hartley. "Improving the quality of natural language requirements specifications through natural language requirements patterns". In: *The Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*. IEEE. 2006, pp. 199–199.

[TMJ12] Mugur Tatar, Jakob Mauss, and Andreas Junghanns. *Systematic Test and Validation of Automotive Systems*. Tech. rep. QTronic GmbH, 2012.

[Töl96] Winfried Tölle. "Ein Fahrmanöverkonzept für einen maschinellen Kopiloten". In: *Fortschritt-Berichte VDI, Reihe 12: Verkehrstechnik/Fahrzeugtechnik* (1996).

[Ulb+14] Simon Ulbrich et al. "Graph-based context representation, environment modeling and information aggregation for automated driving". In: *IEEE Intelligent Vehicles Symposium, Proceedings*. 2014. ISBN: 9781479936380.

[Ulb+15] Simon Ulbrich et al. “Defining and Substantiating the Terms Scene, Situation, and Scenario for Automated Driving”. In: *IEEE 18th International Conference on Intelligent Transportation Systems*. IEEE. 2015, pp. 982–988.

[Ulb+16] Simon Ulbrich et al. “Testing and Validating Tactical Lane Change Behavior Planning for Automated Driving”. In: *Automated Driving - Safer and more efficient future driving*. Ed. by Martin Horn and Daniel Watzenig. (accepted to appear). Springer International Publishing AG, 2016.

[UM13] Simon Ulbrich and Markus Maurer. “Probabilistic online POMDP decision making for lane changes in fully automated driving”. In: *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*. 2013, pp. 2063–2067.

[UM15a] Simon Ulbrich and Markus Maurer. “Situation Assessment in Tactical Lane Change Behavior Planning for Automated Vehicles”. In: *IEEE 18th International Conference on Intelligent Transportation Systems*. 2015, pp. 975–981.

[UM15b] Simon Ulbrich and Markus Maurer. “Towards Tactical Lane Change Behavior Planning for Automated Vehicles”. In: *IEEE 18th International Conference on Intelligent Transportation Systems*. 2015, pp. 989–995.

[US 13] U.S. Department of Transportation, National Highway Traffic Safety Administration. *Preliminary statement of policy concerning automated vehicles*. 2013, pp. 1–14.

[VDI14] VDI - Gesellschaft Produktion und Logistik. *VDI3633: Simulation of systems in materials handling, logistics and production - Fundamentals*. Tech. rep. Verein Deutscher Ingenieure (VDI), 2014.

[VDM95] Axel Van Lamsweerde, Robert Darimont, and Philippe Massonet. “Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt”. In: *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE’95)*. IEEE. 1995, pp. 194–203.

[Ver+00] Lennard Verhoeff et al. “VEHIL: A full-scale test methodology for intelligent transport systems, vehicles and subsystems”. In: *Intelligent Vehicles Symposium*. IEEE. 2000.

[Vis+08] Costandinos Visvikis et al. “Study on lane departure warning and lane change assistant systems”. In: *Transport Research Laboratory Project Rpt PPR* 374 (2008).

[VKV00] Arie Van Deursen, Paul Klint, and Joost Visser. “Domain-specific languages: An annotated bibliography”. In: *ACM Sigplan Notices* 35.6 (2000), pp. 26–36.

[VVP02] Dirk J. Verburg, Albert C. M. Van der Knaap, and Jeroen Ploeg. “VEHIL: developing and testing intelligent vehicles”. In: *Intelligent Vehicle Symposium*. IEEE. 2002.

[Wan+07] Yiqiao Wang et al. “An automated approach to monitoring and diagnosing requirements”. In: *Automated Software Engineering Conference*. ACM. 2007, pp. 293–302.

[Wed16] Sebastian Wedeniwski. *The Mobility Revolution in the Automotive Industry*. Springer, 2016.

[Wei+14] Alexander Weitzel et al. “Absicherungsstrategien für Fahrerassistenzsysteme mit Umfeldwahrnehmung Forschungsbericht der Bundesanstalt für Straßenwesen, Bereich Fahrzeugtechnik”. In: *Berichte der Bundesanstalt für Straßenwesen*. Vol. 98. Wirtschaftsverlag NW, 2014.

[WH06] John Whitelegg and Gary Haq. *Vision Zero: Adopting a Target of Zero for Road Traffic Fatalities and Serious Injuries*. Tech. rep. Stockholm Environment Institute, 2006.

[WH07] Conal Watterson and Donal Heffernan. “Runtime verification and monitoring of embedded systems”. In: *IET Software* 1.5 (Oct. 2007), pp. 172–179. ISSN: 1751-8806.

[WH08] Conal Watterson and Donal Heffernan. “A runtime verification monitoring approach for embedded industrial controllers”. In: *IEEE International Symposium on Industrial Electronics*. IEEE. 2008, pp. 2016–2021.

[Win+18] Hermann Winner et al. “PEGASUS - First Steps for the Safe Introduction of Automated Driving”. In: *Automated Vehicles Symposium 2018*. Springer. 2018, pp. 185–195.

[Win15] Hermann Winner. “ADAS, Quo Vadis?” In: *Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort*. Ed. by Hermann Winner et al. Cham: Springer International Publishing, 2015, pp. 1557–1584. ISBN: 978-3-319-12352-3.

[WS05] David P. Watson and David H. Scheidt. “Autonomous Systems”. In: *Johns Hopkins APLTechnical Digest* 26.4 (2005).

[Wu+17] Meng Wu et al. “Safety guard: Runtime enforcement for safety-critical cyber-physical systems”. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM. 2017, p. 84.

[WW15] Walther Wachenfeld and Hermann Winner. “Virtual Assessment of Automation in Field Operation A New Runtime Validation Method”. In: *10. Workshop Fahrerassistenzsysteme*. 2015, p. 161.

[WW16] Walther Wachenfeld and Hermann Winner. “The Release of Autonomous Vehicles”. In: *Autonomous Driving: Technical, Legal and Social Aspects*. Ed. by Markus Maurer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 425–449. ISBN: 978-3-662-48847-8.

[WW17] Walther Wachenfeld and Hermann Winner. "The New Role of Road Testing for the Safety Validation of Automated Vehicles". In: *Automated Driving: Safer and More Efficient Future Driving*. Ed. by Daniel Watzenig and Martin Horn. Cham: Springer International Publishing, 2017, pp. 419–435. ISBN: 978-3-319-31895-0.

[You01] Ralph R. Young. "Effective requirements practices". In: *Information Technology Series. Addison-Wesley* (2001).

[ZAM14] Xueyi Zou, Rob Alexander, and John McDermid. "Safety validation of sense and avoid algorithms using simulation and evolutionary search". In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2014, pp. 33–48.

[Zei13] Knut Zeißler. "Fahrspurerkennung in LIDAR-Punktwolken". MA thesis. Institut fur Informatik, Arbeitsgruppe Künstliche Intelligenz, 2013.

[Zel+10] Marc Zeller et al. "Co-simulation of self-adaptive automotive embedded systems". In: *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. IEEE. 2010, pp. 73–80.

[Zha16] Ding Zhao. "Accelerated Evaluation of Automated Vehicles." PhD thesis. The University of Michigan, 2016.

[Zof+15] M. R. Zofka et al. "Data-driven simulation and parametrization of traffic scenarios for the development of advanced driver assistance systems". In: *18th International Conference on Information Fusion*. 2015, pp. 1422–1428.